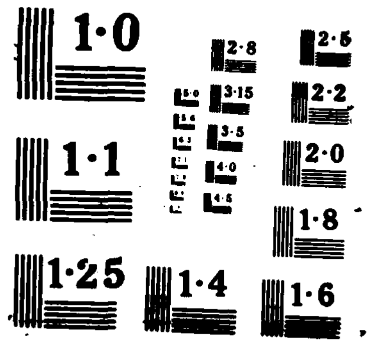
AD-A176 114 AN EFFICIENT DISTRIBUTED ALGORITHM FOR MAXIMUM MATCHING 1/1
IN GENERAL GRAPHS(U) ILLINOIS UNIV AT URBANA APPLIED
COMPUTATION THEORY GROUP M C WU JAN 87 ACT-73
UNCLASSIFIED N00014-85-K-0570 F/G 9/2 NL


1·0
2·8
2·5
3·15
2·2
3·5
1·1
4·0
2·0
4·5
1·8
1·25
1·4
1·6

January 1987

UILU-ENG-87-2201
ACT-73

COORDINATED SCIENCE LABORATORY
College of Engineering
Applied Computation Theory

12

AN EFFICIENT DISTRIBUTED ALGORITHM FOR MAXIMUM MATCHING IN GENERAL GRAPHS

Michael C. Wu

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-87-2201 ACT-73	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois; 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State and ZIP Code) 1101 W. Springfield Avenue Urbana, Illinois 61801	7b. ADDRESS (City, State and ZIP Code) 800 N. Quincy Street Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research; 8b. OFFICE SYMBOL (If applicable) N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-85-K-0570
8c. ADDRESS (City, State and ZIP Code) 800 N. Quincy Street Arlington, VA 22217	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. N/A; PROJECT NO. N/A; TASK NO. N/A; WORK UNIT NO. N/A
11. TITLE (Include Security Classification) An Efficient Distributed Algorithm for Maximum Matching in General Graphs	
12. PERSONAL AUTHOR(S) Michael C. Wu	
13a. TYPE OF REPORT Technical; 13b. TIME COVERED FROM ____ TO ____	14. DATE OF REPORT (Yr., Mo., Day) January 1987; 15. PAGE COUNT 63
16. SUPPLEMENTARY NOTATION N/A	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) graph, matching, combinatorial optimization, distributed algorithm

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We present a distributed algorithm for maximum cardinality matching in general graphs. In the worst case the algorithm uses $O(|V|^{5/2})$ messages. On trees the algorithm uses only $O(|V|)$ messages.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code); 22c. OFFICE SYMBOL NONE

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE. Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

AN EFFICIENT DISTRIBUTED ALGORITHM FOR MAXIMUM MATCHING IN GENERAL GRAPHS

BY

MICHAEL M. WU

B.S., University of Illinois, 1985

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1987

Urbana, Illinois

ACKNOWLEDGMENTS

First, I would like to thank my advisor Michael Loui. His suggestions, insight, and guidance were invaluable in the completion of this thesis. I would also like to thank my family for their support. Finally, I thank my friends for their words of encouragement.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 The Problem

Let $G = (V, E)$ be a finite, undirected, connected graph with the set of vertices V and the set of edges E. A *matching* M is a subset of E such that no two edges of M are incident on a common vertex. A *maximum matching* is a matching of maximum cardinality. ~~We~~ presents an efficient distributed algorithm for finding a maximum matching in a general graph.

1.2 Previous Work

The maximum matching problem is a fundamental problem of combinatorial optimization and has been extensively studied. We summarize some of the previous work done in maximum matching algorithms and distributed algorithms.

1.2.1 Sequential matching algorithms

For maximum matching in bipartite graphs, the algorithm of Hopcroft and Karp (1973) is the fastest known with a running time of $O(|V|^{1/2}|E|)$.

A more difficult problem is computing maximum matchings in general graphs. Among the sequential algorithms that have been proposed for finding maximum matchings in general graphs are those of Edmonds (1965), Kameda and Munro (1974), Even and Kariv (1975), Gabow (1976), and Micali and Vazirani (1980). For general graphs, the algorithm of Micali and Vazirani is the most efficient known with a running time of $O(|V|^{1/2}|E|)$.

We give an overview of the algorithm of Micali and Vazirani. Peterson (1985) gave an exposition of their algorithm. To find a maximum matching, the algorithm proceeds in phases. During each phase, the algorithm finds a maximal set of vertex disjoint minimum length augmenting paths and increases the matching along these paths. Hopcroft and Karp (1973) proved that $O(|V|^{1/2})$

such phases suffice to find a maximum matching. Each phase of the algorithm runs in $O(|E|)$ time. Thus, the running time of the algorithm is $O(|V|^{1/2}|E|)$.

1.2.2 *Distributed matching algorithms*

Recently, Schieber and Moran (1986) presented a distributed algorithm for finding maximum matchings in general graphs. No efficient distributed algorithms for the maximum matching problem was known before. Their algorithm runs in time $O(|V| \log |V|)$, assuming all the processors are synchronized, internal processing takes zero time, and each message arrives at its destination exactly one time unit after it has been sent. The communication complexity of their algorithm depends upon the model. In the memory restricted model, where the amount of storage at each vertex is bounded by a linear function of its degree, the communication complexity is $O(|V|^2|E|)$ messages. If the amount of storage at each vertex is unrestricted, then the communication complexity is $O(|V||E| \log |V|)$ messages.

Their observation is that, unlike the sequential case, in a distributed network, a search for a single minimum length augmenting path can be made faster than a search for a maximal set of minimum length augmenting paths. This is because in the sequential case $O(|E|)$ time is needed to find either one augmenting path or a maximal set of such paths. In the distributed case, however, the search for augmenting paths can be made in parallel. Thus an augmenting path of length l can be found in $O(l)$ time, whereas $O(|V|)$ time would be required to find a maximal set of such paths. Using this observation, they showed that $O(|V|)$ iterations of finding one minimum length augmenting path are faster than $O(|V|^{1/2})$ iterations of finding a maximal set of such augmenting paths.

1.2.3 *A technique for designing distributed algorithms*

Awerbuch (1985a) presented a technique called a *synchronizer* for designing efficient distributed algorithms in asynchronous networks. The synchronizer allows algorithms for asynchronous networks to be designed as if they were to be executed on a synchronous network. The motivation behind using a synchronizer is that asynchronous algorithms often have time or communication complexities much worse than their corresponding synchronous algorithms. Thus, if the additional complexity introduced by the synchronizer is small as compared with the complexity of the synchronous algorithm, then an efficient distributed algorithm can be obtained by adding a synchronizer to the synchronous algorithm.

Awerbuch demonstrated the power of the synchronizer on distributed algorithms for breadth first search and maximum flow. For breadth first search, a previous distributed algorithm by Gallgher (1982) has a communica[illegible]nplexity of $|V||E|$ messages and a time complexity of $|V|$. The distributed algor[illegible]ained by using a synchronizer with a parallel breadth first search algorithm by Eckstein (1977) improved the communication complexity to $k|V|^2$ messages, where k is a parameter, $2 \leqslant k \leqslant |V|$. The time complexity of the algorithm is $|V| \dfrac{\log_2|V|}{\log_2 k}$.

For maximum flow, Segall (1982) presented an algorithm with a message complexity of $|V||E|^2$ and a time complexity of $|V|^2|E|$. The algorithm obtained by using a synchronizer with the parallel maximum flow algorithm by Shiloach and Vishkin (1982) has a message complexity of $k|V|^3$ and a time complexity of $|V|^2 \dfrac{\log_2|V|}{\log_2 k}$.

1.3 Our Distributed Matching Algorithm

In designing ~~our~~ distributed matching algorithm, ~~we are~~ mainly concerned with the communication complexity. The maximum number of message transmissions determines the efficiency of the algorithm. ~~We~~ the concentrates on minimizing the number of messages for two reasons. First, in an actual distributed system, the communication time would likely be much greater than the processing time. Second, in commercial computer networks, common carriers often charge by the number of packets or bits rather than by time.

We designed our distributed matching algorithm by integrating Awerbuch's synchronizer technique with the sequential matching algorithm of Micali and Vazirani. We could not directly make a synchronized implementation of the algorithm of Micali and Vazirani, however, because of differences in the characteristics of distributed and shared-memory networks. Also, we significantly improved the efficiency of the distributed algorithm by modifying the straightforward implementation of certain procedures of the sequential algorithm. The communication complexity of our algorithm is $O(|V|^{5/2})$ messages with bounded storage at each vertex. Since the graph is connected, $|E| \geqslant |V| - 1$. Thus, the communication complexity of our algorithm is better than that of the algorithm of Schieber and Moran. Our algorithm, however, searches for augmenting paths sequentially. Thus, the time complexity of our algorithm can be as large as the message complexity, $O(|V|^{5/2})$.

CHAPTER 2

DEFINITIONS

2.1 The Model of Computation

We present our model of distributed computation. We first give a general description of the model and then give some precise definitions.

2.1.1 Overview

The distributed computation model is an asynchronous network described by an undirected, finite, connected graph $G = (V, E)$ with the set of vertices V and the set of edges E. The set of vertices V represents the processors of the network, and the set of edges E represents the bidirectional communication links between the processors. Thus, the network topology determines the graph. In our discussion, we will refer to the processors as vertices and the links as edges. An *edge* (x, y) means that there is a link connecting processors x and y.

A vertex x is a *neighbor* of vertex y if there is an edge (x, y) in E. Each vertex has a distinct identity and knows the identities of its neighbors. Neighboring vertices can send messages to each other along the same edge in both directions simultaneously. A vertex can send messages to more than one neighbor simultaneously but can receive messages only one at a time. The edges are assumed to be error free so messages arrive in sequence without error after an unpredictable but finite delay.

Local computations are assumed to require negligible time. We make this assumption because in an actual system, the communication time would likely be much greater than the processing time. Each vertex receives messages from its neighbors, performs local computations, and sends messages to its neighbors.

The communication complexity of a distributed algorithm is the maximum possible number of messages all vertices of the distributed network may send during the execution of the algorithm.

The time complexity of the algorithm is the maximum possible execution time of the algorithm assuming a message sent by the source requires exactly one time unit to arrive at its destination. This model appears in Awerbuch (1985a), Schieber and Moran (1986), and others.

We use the memory restricted model of Schieber and Moran (1986), where the amount of storage at each vertex is bounded by a linear function of its degree. The length of each message is proportional to log $|V|$.

Note that if both the message length and the amount of storage at each vertex are unrestricted, then after a spanning tree is constructed, only $O(|V|)$ messages are needed to find a maximum matching. Since the amount of storage at each vertex is unrestricted, we send all the information about the network topology to the root. Then the root computes a maximum matching and sends the result to the rest of the network.

To convey the network topology to the root, each vertex sends a message to its parent along the spanning tree. A vertex v sends a message which gives the neighbors of v and the neighbors of each of the descendants of v. After the root receives a message from each of its children in the spanning tree, the root knows the topology of the network and computes a maximum matching. Then the root sends a message which gives the solution to each of its children in the spanning tree. A vertex receiving the solution sends it its children in the spanning tree.

If the amount of storage at each vertex is unrestricted, but the message length is restricted, then after a spanning tree is constructed, $O(|V||E|)$ messages are needed to find a maximum matching. Again we send all the information about the network topology to the root along the spanning tree, and the root computes the maximum matching. The method is the same as before except that each message gives the information about only one edge, i.e., one pair of neighbors. Since there are $|E|$ edges, there are $O(|E|)$ messages. Since the vertices send the messages along the spanning tree, the total number of messages is $O(|V||E|)$. This justifies restricting both the message length and the amount of storage at each vertex.

2.1.2 Definitions

We now give a precise definition of the distributed computation model. Our definition is similar to the one presented in Gafni *et al* (1984). A *distributed system* is a triple (PROCS, LINKS, MSGS), where

PROCS is a finite set of vertices

LINKS $\subseteq$ PROCS $\times$ PROCS is a set of edges

MSGS is a set of messages.

An *event* is the transmission or arrival of a message at a vertex x. An event is specified by giving the vertex, the edge, the message, and whether the event is a transmission or arrival.

Each vertex has a current state. The *state* of a vertex x is specified by a sequence of zero or more events at x. A state of a vertex x has the form

$$<e_1, e_2, \cdots, e_l>,$$

where each event e_i is an arrival of a message at x or the transmission of a message by x. Note that the state of a vertex does not depend upon the time between events.

Let s be the current state of x. When an event e occurs at x, x makes a transition from the current state s to a new state by concatenating e onto the end of s. Let STATES be the set of states.

A *configuration* is a function C that specifies a state for each vertex and the messages on each edge. Each edge has either zero or one message in transit in each direction. If all vertices have the empty state ϵ and no messages are in transit, then the configuration is *initial*.

A *distributed algorithm* is a function

$$A : \text{STATES} \rightarrow (\text{MSGS} \times E) \cup \{\emptyset\}$$

that specifies what a vertex does in any state. If a vertex x is in state s, then x either sends a message to another vertex as specified by $A(R(s)) \in$ MSGS $\times$ E and changes state or does nothing

$(A(s) = \emptyset)$.

An *execution* of an algorithm A is a finite sequence of configurations

$$C_0, C_1, \cdots, C_f,$$

starting from an initial configuration C_0 such that for all i, C_{i+1} is obtained from C_i by either the transmission or arrival of a message at some vertex x. An execution *terminates* in a final configuration C_f when every vertex of the system is in a state in which it does nothing and no messages are in transit.

The *message complexity* of an algorithm A is a function $f(|V|, |E|)$ that gives the maximum number of messages sent in executions of A on a distributed system with $|V|$ vertices and $|E|$ edges. The *time complexity* of an algorithm A is a function $g(|V|, |E|)$ that gives the maximum amount of time required for executions of A on a distributed system with $|V|$ vertices and $|E|$ edges assuming that each message arrives at its destination exactly one time unit after it has been sent.

2.2 Matching Definitions

Let M be a matching in G. The following terms are defined for a fixed matching M. An edge e is *matched* if $e \in M$ and *free* if $e \notin M$. A vertex v is *matched* if a matched edge is incident on v and *exposed* if no matched edges are incident on v. If the edge (v, w) is matched, then v is the *mate* of w and vice versa.

In Figure 1, straight lines represent the free edges and dotted lines the matched edges. The free edges are (A, C), (B, C), (D, E), and (D, F). The edge (C, D) is a matched edge. Vertices A, B, E, and F are exposed and vertices C and D are matched. Thus C is the mate of D and vice versa.

An *alternating path* is a path $(v_1, v_2, \cdots)$ whose edges $(v_1, v_2), (v_2, v_3), \cdots$ are alternately in M and not in M. An *augmenting path* is an alternating path whose first and last vertices are exposed. In Figure 1, the path (A, C, D, F) is an augmenting path. Two paths are *disjoint* if they

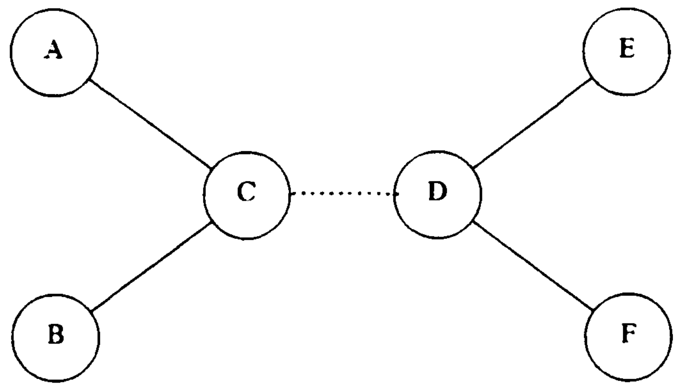


Figure 1. A Matching M

have no common vertices.

The *evenlevel* of a vertex v is the length of the minimum even length alternating path leading from v to an exposed vertex, if any, and infinite otherwise. The *oddlevel* of a vertex v is the length of the minimum odd length alternating path leading from v to an exposed vertex, if any, and infinite otherwise. The *level* of a vertex v is the smaller of evenlevel (v) and oddlevel (v). Thus level (v) is the length of the minimum length alternating path leading from v to an exposed vertex. A vertex v is *outer* if level (v) is even and *inner* if level (v) is odd. If v is outer, then the *otherlevel* of v is oddlevel (v) and vice versa.

A *blossom* is a circuit of odd length that is maximally matched. A blossom with length $2k + 1$ has k matched edges. Figure 2 contains the blossom (C, D, E, F, G).

An edge (v, w) is a *bridge* if either both evenlevel (v) and evenlevel (w) are finite or both oddlevel (v) and oddlevel (w) are finite. The discovery of a bridge signifies the presence of either an augmenting path or a blossom. The *tenacity* of a bridge (v, w) = min {evenlevel (v) + evenlevel (w), oddlevel (v) + oddlevel (w)} + 1. If there are no blossoms, then the tenacity of bridge (v, w) is the length of the minimum length augmenting path containing bridge (v, w).

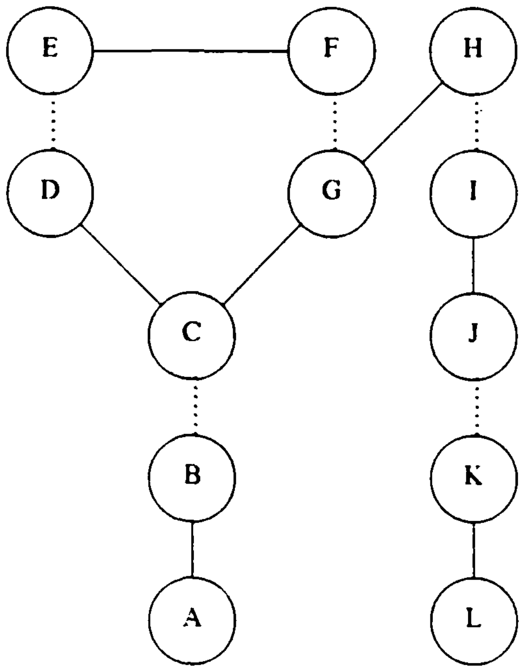


Figure 2. A Blossom

A vertex u is an *anomaly* of a vertex v if v is inner, u is outer, u and v are neighbors, and evenlevel $(u) > $ oddlevel (v). In Figure 2, H is an anomaly of G since evenlevel (H) = 4 and oddlevel (G) = 3.

We present two theorems about augmenting paths.

Theorem 2.1: Let P be the edges of an augmenting path with respect to a matching M in a graph G. Let $M' = M \oplus P$ be the matching with the set of edges e such that either $e \in M$ and $e \notin P$ or $e \notin M$ and $e \in P$. Then M' is a matching of cardinality $|M| + 1$.

The following theorem is due to Berge (1957):

Berge's Theorem: A matching M in a graph G is maximum if and only if there is no augmenting path in G with respect to M.

Papadimitriou and Steiglitz (1982) gave clear proofs of these theorems.

We define *increasing the matching along P* to be the process of converting the matching M into the matching M' as follows. We *reverse* the matching of an edge by making a matched edge free, and vice versa. The matching M is converted into the matching M' by reversing the matching of the edges in P. In Figure 3a (Peterson, 1985) is a matching M. The result of increasing the matching along the augmenting path (C, D, E, F) is the matching M' shown in Figure 3b.

2.3 Sending Messages

We give some definitions for sending messages. When a vertex v *forwards* a message MSG to a vertex w, v sends a message MSG to w identical to the one that v received. We now describe

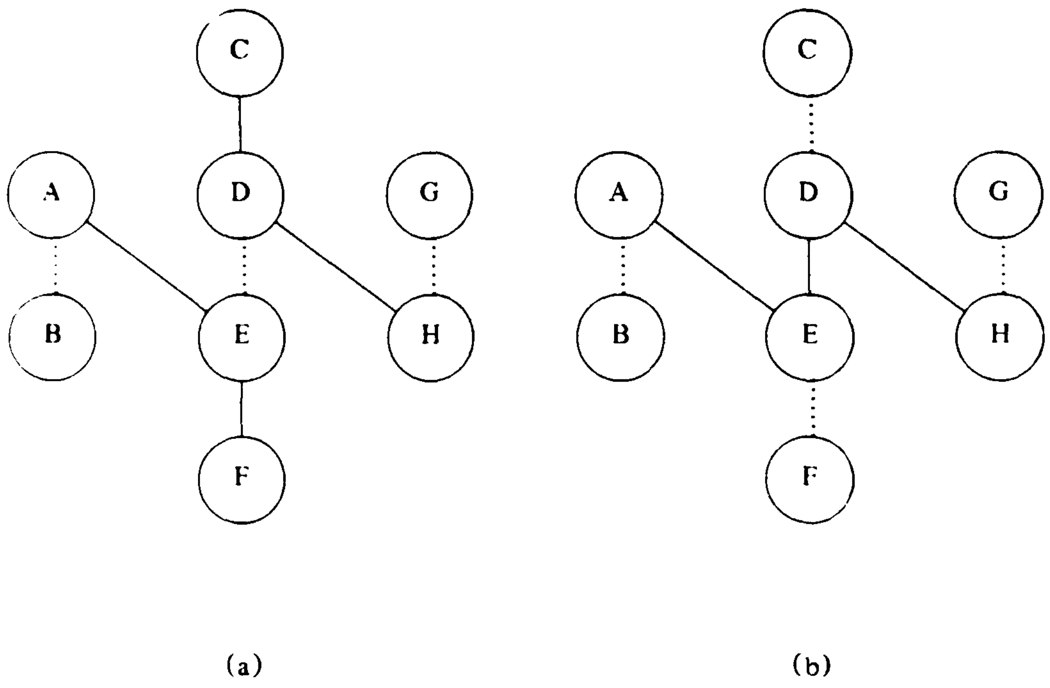


Figure 3. Increasing the Matching

two message passing paradigms.

Suppose we have a spanning tree T of the network. A *broadcast* is a communication pattern in which the root of T sends a message MSG and all vertices of T eventually receive MSG. To perform the broadcast, each vertex v receiving a message MSG forwards MSG to each of v's spanning tree children. The broadcast is initiated by the root and terminates at the leaves of T.

A *convergecast* is a communication pattern in which the root receives a message from a child in T only if all descendants of the child have sent convergecast messages. A convergecast can be thought of as being the opposite of a broadcast. A vertex v sends a convergecast message only after all descendants of v have sent their convergecast messages. The messages in the convergecast, however, need not be the same. To perform the convergecast, a vertex v sends a convergecast message to its parent in T after v has received a convergecast message from each of its children in T. The convergecast begins at the leaves of T, which have no children, and terminates at the root.

Suppose we have the spanning tree shown in Figure 4. To perform a broadcast of a message MSG, vertex A sends message MSG to vertices B and C. B forwards MSG to D and C forwards MSG to E, F, and G. To perform a convergecast, vertices D, E, F, and G send convergecast messages

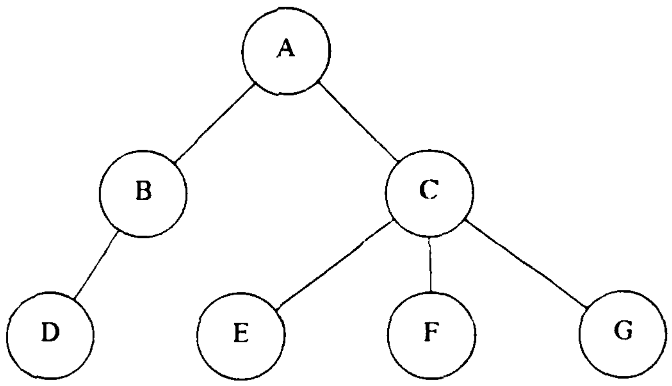


Figure 4. A Spanning Tree

to their parents in T. B sends a convergecast message to A after receiving a convergecast message from its child D. and C sends a convergecast message to A after receiving convergecast messages from its children E. F. and G.

CHAPTER 3

THE DISTRIBUTED MATCHING ALGORITHM

3.1 Preprocessing

To implement Awerbuch's synchronizer technique, we need to construct a synchronization network. Since the vertices synchronize by sending messages to each other, we minimize the number of messages by using a spanning tree for the synchronization network. We select the root of the spanning tree to be the leader of the network.

The leader synchronizes the various steps of the algorithm via broadcasts and convergecasts. The leader begins a step by performing a broadcast and recognizes the end of the step through a convergecast.

In our algorithm, we are not concerned about the construction of the synchronization tree. Thus we will assume that a spanning tree ST of the network has been constructed and the root selected to be the leader of the network. The *ST-parent* of a vertex v is the parent of v in the spanning tree ST. Similarly, the *ST-children* of v are the children of v in ST and the *ST-descendants* of v are the descendants of v in ST.

There are distributed algorithms for constructing spanning trees. Gallagher, Humblet, and Spira (1983) presented a distributed algorithm for constructing minimum weight spanning trees requiring at most $5|V| \log |V| + 2|E|$ messages. Note that we do not need the minimum weight. Their algorithm maintains a distinguished edge called the core to initiate the cycles of the algorithm. When the spanning tree is found, we can choose a vertex of the core to be the leader.

Awerbuch (1985b) presented a distributed algorithm for constructing a depth first search spanning tree which uses at most $4|E|$ messages. The algorithm begins at the root and adds vertices to the tree until all the vertices have been added.

3.2. Vertex Description

We give a description of the vertices representing the processors of the network. We use the memory restricted model of Schieber and Moran (1986) where the amount of storage at each vertex is bounded by a linear function of its degree. We allow a vertex a fixed amount of storage for each incident edge. This is a reasonable assumption since a vertex with a larger number of edges requires a proportionately larger amount of storage for variables and buffers. The vertices have unique identities but are otherwise identical. Every vertex executes the same algorithm with the exception that the leader must also synchronize the remaining vertices. The algorithm at each vertex can easily be made the same, however, if we include a test for the leader.

The following is a description of the local variables maintained by each vertex v. The functions of some of the variables will be further explained as they are used.

evenlevel (v)	evenlevel of v
oddlevel (v)	oddlevel of v
level (v)	minimum (evenlevel (v), oddlevel (v))
neighbors (v)	the neighbors of v
erased (v)	true if v is erased
activeneighbors (v)	the neighbors of v that are not erased
predecessors (v)	the predecessors of v
visited (v)	the bridge vertex that last visited v during the phase
mate (v)	if v is matched, the mate of v
bridges (v)	the vertices that form a bridge with v
anomalies (v)	the anomalies of v
blossom (v)	if v belongs to a blossom, the base of the blossom
stparent (v)	the ST-parent of v
stchildren (v)	the ST-children of v
dfsparent (v)	the depth first search parent of v
dfschild (v)	the depth first search child of v
dcvchild (v)	the child of v along the path to the deepest common vertex
counter (v)	message counter
barrier (v)	used to prevent redundant backtracking beyond v
altpath (v)	true if v is on an alternating path to an exposed vertex

We assume that neighbors (v), stparent (v), and stchildren (v) are set during preprocessing. Their values remain the same throughout the algorithm.

3.3 Algorithm Overview

We give a high-level overview of the algorithm. To find a maximum matching, the algorithm proceeds in phases. During each phase, the algorithm finds a maximal set of disjoint minimum length augmenting paths and increases the matching along those paths. Note that during each phase we need to find only a maximal set of such paths, not a maximum set. Hopcroft and Karp (1973) proved that $O(|V|^{1/2})$ such phases suffice to find a maximum matching. The phases are numbered 0, 1, 2, ...

Next, we describe the execution of one phase of the algorithm. The objective of each phase is to find a maximal set of minimum length augmenting paths. To find augmenting paths, the algorithm performs breadth first search from exposed vertices to find bridges. If any bridges are found, the algorithm tries to find augmenting paths containing each bridge one at a time. The algorithm increases the matching as each augmenting path is found. If any augmenting paths are found during the current phase, then a maximal set of minimum length augmenting paths is found, and the algorithm proceeds to the next phase. If no augmenting paths are found, then the matching is maximum (Berge, 1957).

To find bridges, the algorithm performs breadth first search from exposed vertices. The search proceeds one level at a time. The search levels are numbered 0, 1, 2, ... The initial search level of each phase is 0. When the current search level is i, a search is made for all bridges at level i. If a bridge is discovered, then the algorithm performs depth first search from the bridge vertices to find an augmenting path. If no augmenting paths containing bridges at the current search level are found, then the breadth first search proceeds to the next level. If an augmenting path is found, then the algorithm begins a new phase.

To find an augmenting path containing a bridge, the algorithm performs depth first search from the bridge vertices of the bridge to find alternating paths to exposed vertices. The algorithm searches for augmenting paths one at a time. If the algorithm finds two disjoint alternating paths leading from the vertices of a bridge to exposed vertices, then the algorithm increases the matching along the augmenting path consisting of the two alternating paths and the bridge. To increase the matching, the algorithm reverses the matching of the edges along the augmenting path. If two disjoint alternating paths from the bridge vertices to exposed vertices cannot be found, then there is no augmenting path. There may, however, be a blossom.

The algorithm continues to increase the matching until a maximum matching is obtained. If the matching is maximum, then there are no augmenting paths (Berge, 1957). The algorithm recognizes that the matching is maximum and halts when the breadth first search for bridges reaches a level i such that no vertices are at level i.

We now describe the matching algorithm in detail. We examine how the algorithm proceeds during a phase p.

3.4 Phase Initialization

The leader L starts a phase p by broadcasting a STARTPHASE (p) message. When a vertex v receives a STARTPHASE (p) message, v forwards the message to its ST-children and initializes its local variables as follows:

```
evenlevel (v) := ∞
oddlevel (v):= ∞
level (v) := ∞
predecessors (v) := Ø
activeneighbors (v) := neighbors (v)
anomalies (v) := Ø
visited (v) := nil
erased (v) := false
bridges (v) := Ø
barrier (v) := nil
altpath (v) := false
dfsparent (v) := nil
dfschild (v) := nil
dcvchild (v) := nil
if phase p = 0 then mate (v) := nil
if v is exposed then evenlevel (v) := 0, level (v) := 0
```

During the algorithm, once a vertex v becomes matched, v remains matched throughout the remainder of the algorithm. But v may be matched with a different vertex during different phases

Since the augmenting paths found during each phase are disjoint, the mate of a vertex changes at most once per phase.

Vertex v convergecasts a READY message after v has initialized its variables and received a READY message from each of its ST-children. The phase initialization is synchronized via the STARTPHASE and READY messages.

3.5 Search for Bridges

We describe the search for bridges at level i. For now we consider the case where there are no blossoms. This will simplify the discussion. We examine the general case with blossoms in Section 3.10.

To search for bridges, the algorithm performs breadth first search from exposed vertices. To perform the search, after L receives a READY message from each of its ST-children, L sets the current search level i and broadcasts a STARTBFS (i) message. The initial search level for each phase is 0. The STARTBFS (i) message signals vertices at level i to extend breadth first search to level $i+1$. Thus when $i = 0$, exposed vertices z start breadth first search since they are the only ones with level $(z) = 0$. During the first phase, all vertices are exposed.

A vertex v at level i receiving a STARTBFS (i) message searches for bridges at level i. Vertex v continues the search along alternating paths to those vertices that are unsearched and unerased. Let u be the vertices in activeneighbors (v) - predecessors (v). Then the vertices u are unsearched and unerased. To continue the search along an alternating path, if i is even, then v sends BRIDGESEARCH $(v, i+1)$ messages to those vertices u such that the edge (v, u) is free. If i is odd, then v sends BRIDGESEARCH $(v, i+1)$ messages to those vertices u such that the edge (u, v) is matched. Vertex v sets counter (v) to the number of BRIDGESEARCH messages it sent.

A vertex u receiving a BRIDGESEARCH (v, j) message, where j is $i+1$, proceeds as follows. If j is even and evenlevel $(u) = \infty$, then u sets evenlevel $(u) := j$ and adds v to predecessors (u). Similarly, if j is odd and oddlevel $(u) = \infty$, then u sets oddlevel $(u) := j$ and adds v to predecessors (u). If necessary, u updates level (u). The predecessors of u are the vertices v at level $(u) - 1$.

If j is odd and $j >$ oddlevel (u), then v is an anomaly of u since evenlevel $(v) \geqslant$ oddlevel $(u) + 1$. Thus u adds v to anomalies (u).

Note that if vertex u receives more than one BRIDGESEARCH message at the current search level, then all the BRIDGESEARCH messages will have the same j. Also note that the search for bridges discovers the vertices at the next search level. Thus if no augmenting path is discovered at the current search level i, then the vertices at search level $i + 1$ will already have been identified, and L need only broadcast a STARTBFS $(i + 1)$ message to continue with the next level of the breadth first search.

After u receives the BRIDGESEARCH (v, j) message, u determines the edge (u, v) is a bridge if either j is odd and evenlevel (u) is finite or if j is even and oddlevel (u) is finite. In the case without blossoms, if the edge (u, v) is a bridge, then level $(u) = j - 1$ since level $(u) =$ level (v). Note that for each pair of neighbors u and v that form a bridge, each vertex sends a BRIDGESEARCH message to the other.

If u determines that the edge (u, v) is a bridge, then u adds v to bridges (u). Vertex u deletes v from predecessors (u) since v is at the same level as u. Then u sends a BRIDGEREPLY $(u$, **bridge**$)$ message to v to inform v that the edge (u, v) is a bridge. If u determines that the edge (u, v) is not a bridge, then u sends a BRIDGEREPLY $(u$, **nobridge**$)$ message to v. Vertex u sends a BRIDGEREPLY message to each neighbor from which it received a BRIDGESEARCH message.

A vertex v receiving a BRIDGEREPLY $(u$, **bridge**$)$ message adds u to bridges (v) and decrements counter (v) by 1. A vertex v receiving a BRIDGEREPLY $(u$, **nobridge**$)$ message just decrements counter (v). The variable counter (v) maintains the number of neighbors to which v sent BRIDGESEARCH messages that have not returned BRIDGEREPLY messages. Vertex v continues to receive BRIDGEREPLY messages until counter $(v) = 0$.

All vertices v perform a convergecast to inform the leader whether any bridges were discovered. For the convergecast, each vertex sends one of two types of messages. A vertex v sends a BRIDGES (v) message if either v or some ST-descendant of v discovered a bridge. Otherwise, v sends a NOBRIDGES (v, *atlevel*) message, where the boolean *atlevel* is used to inform the leader whether the algorithm should halt. If v or some ST-descendant of v is at level i, then *atlevel* = **true**.

We describe the convergecast and the computation of *atlevel* for vertex v. After v has received a BRIDGES or NOBRIDGES message from each of its ST-children and counter (v) = 0, v sends a BRIDGES or NOBRIDGES message as follows. If v received a BRIDGES message from some ST-child of v or if v discovered a bridge, then v sends a BRIDGES (v) message to stparent (v). Otherwise, v sends a NOBRIDGES (v, *atlevel*) message to stparent (v). The value of *atlevel* is set as follows. If level (v) = i or if v received a NOBRIDGES message with *atlevel* = **true** from some ST-child of v, then v sets *atlevel* := **true**. Otherwise, v sets *atlevel* := **false**.

The convergecast of the BRIDGES and NOBRIDGES messages synchronizes the end of the search for bridges at the current search level. When L has received replies from all of its ST-children, L knows whether there were any vertices at the current search level and whether any bridges were discovered.

If L received a BRIDGES message, then the algorithm searches for augmenting paths containing the bridges found at level i. We describe how the algorithm finds augmenting paths in Section 3.6. If an augmenting path is found, then L begins a new phase by incrementing the phase $p := p + 1$ and broadcasting a STARTPHASE (p) message. If no augmenting paths are found, then L continues the search for bridges by incrementing the search level $i := i + 1$ and broadcasting a STARTBFS (i) message.

If L did not receive a BRIDGES message, but did receive a NOBRIDGE message with *atlevel* = **true**, then L increments the search level and continues the breadth first search.

If L receives only NOBRIDGES messages with *atlevel* = **false**, and level $(L) < i$, then the algorithm halts since there are no vertices at level i.

Note that when an anomaly v sends a BRIDGESEARCH message to a vertex u, there is the possibility that the BRIDGESEARCH message from v arrives at u after u has already sent u's BRIDGES or NOBRIDGES message. This occurs if u receives replies from all of its ST-children before the BRIDGESEARCH message from v arrives. But since u sends a BRIDGEREPLY (u, **nobridges**) message to u, the BRIDGES or NOBRIDGES message u sends to stparent (u) is the same message u would have sent if u had not received a BRIDGESEARCH message from v. Note that u does not send a BRIDGESEARCH message to v since from u's point of view, the edge (u, v) is not along an alternating path.

Although u may receive a BRIDGESEARCH message from v after u has sent u's BRIDGES or NOBRIDGES message, the algorithm remains synchronized. Since the leader cannot continue on to the next step of the algorithm until all vertices have convergecasted a BRIDGES or NOBRIDGES message, v convergecasts a BRIDGES or NOBRIDGES message only after v receives a BRIDGEREPLY message from u.

Now we give an example of searching for bridges. Suppose the algorithm is searching for bridges in the part of the network shown in Figure 5. When L broadcasts a STARTBFS (0) message, the exposed vertices C, F, and K begin breadth first search. C sends a BRIDGESEARCH (C, 1) message to B, F sends BRIDGESEARCH (F, 1) messages to E and H, and K sends a BRIDGESEARCH (K, 1) message to J. After the search for bridges at level 1 has been completed, the vertices have the (evenlevel, oddlevel) values shown. When L broadcasts a STARTBFS (2) message, A sends a BRIDGESEARCH (A, 3) message to D and D sends a BRIDGESEARCH (D, 3) message to A. A and D determine that the edge (A, D) is a bridge. G and I find that the edge (G, I) is a bridge. Thus A, D, G, and I convergecast BRIDGES messages and the leader knows that a bridge has been discovered at search level 2.

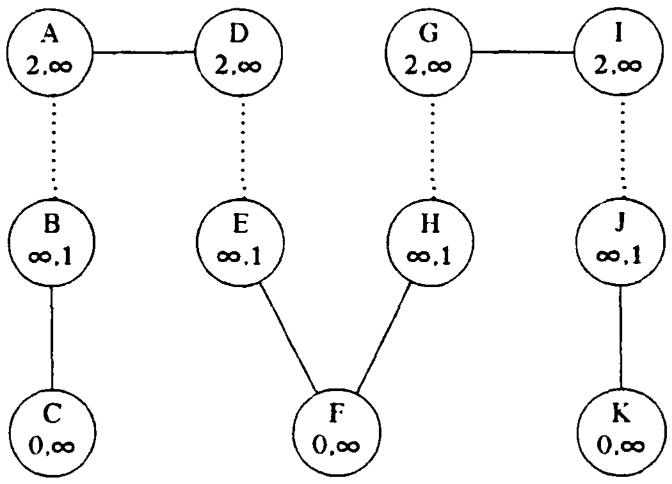


Figure 5. Search for Bridges

3.6 Finding Augmenting Paths

We give an overview of the process of finding augmenting paths. If any bridges are discovered at the current search level, then the algorithm attempts to find augmenting paths containing those bridges. The algorithm searches for augmenting paths one at a time. This ensures that the augmenting paths discovered are disjoint.

To search for augmenting paths one at a time, we consider the vertices of the network in depth first search order on the spanning tree. For our discussion, assume that we order the ST-children of each vertex from left to right with respect to the root of the spanning tree. This simplifies the description of finding augmenting paths. In an actual implementation, the ST-children may be ordered by their identities.

For each bridge discovered at the current search level, the algorithm performs depth first search from the bridge vertices to find alternating paths to exposed vertices. We describe how to find an alternating path from a bridge vertex to an exposed vertex in Section 3.7. If the algorithm is able to find two disjoint alternating paths to exposed vertices, then the algorithm finds an augmenting path formed by the two alternating paths and the bridge. The algorithm increases the matching along the augmenting path and erases the vertices path. Since erased vertices are not considered during the search for alternating paths to exposed vertices, erasing the vertices of an augmenting path ensures the disjointness of subsequent augmenting paths.

Before we describe the execution of the algorithm, we give a definition. If v is a vertex of a bridge (v, w), then v *owns* the bridge (v, w). Thus each bridge is owned by two vertices.

The algorithm considers the vertices of the network in depth first search order from left to right. To begin the process of finding augmenting paths, L sends a STARTAUGMENT message to L's leftmost ST-child. A vertex u receiving a STARTAUGMENT message forwards the message to u's leftmost ST-child, if one exists. The STARTAUGMENT message is forwarded until it reaches a vertex l_0 that has no ST-children, i.e., a leaf.

When l_0 receives the STARTAUGMENT message and finds that it has no ST-children, l_0 knows the algorithm is trying to find an augmenting path containing a bridge owned by l_0.

If l_0 is erased, then l_0 sends a NOTAUGMENTED message to stparent (l_0). If l_0 is not erased, then l_0 checks bridges (l_0) for the bridges that it found. If l_0 did not find any bridges, then l_0 sends a NOTAUGMENTED message to stparent (l_0). If l_0 found at least one bridge, then l_0 arbitrarily chooses a bridge formed with a vertex r_0 in bridges (l_0). Vertex r_0 is the *buddy* of l_0, and vice versa. Then l_0 tries to find an alternating path to an exposed vertex. We describe how a bridge vertex searches for an alternating path to an exposed vertex in Section 3.7.

If l_0 is unable to find an alternating path to an exposed vertex, then there is no augmenting path containing l_0. Vertex l_0 should not be further considered when searching for other augmenting paths during the current phase. Thus l_0 sets erased (l_0) := **true**. Then l_0 sends a NOTAUGMENTED message to stparent (l_0).

If l_0 is able to find an alternating path P to an exposed vertex, then l_0 signals r_0 to search for an exposed vertex by sending r_0 a GO (l_0) message.

For now we will assume that l_0 and r_0 do not encounter any common vertices when searching for alternating paths leading to exposed vertices. If the search for an alternating path encounters an erased vertex, then the search must backtrack and try to find a different path. All vertices are "unerased" at the start of each phase. We consider common vertices and backtracking in Section 3.8.

When r_0 receives the GO (l_0) message, r_0 knows that l_0 is its buddy and that l_0 found an alternating path to an exposed vertex. Vertex r_0 then performs depth first search to find an exposed vertex.

If r_0 is able to find an alternating path P_r to an exposed vertex disjoint from P_l, then there is an augmenting path P_l, (l_0, r_0), P_r. Thus r_0 sends a SUCCESS message to l_0. After l_0 and r_0 increase the matching along the augmenting path, l_0 sends an AUGMENTED message to stparent (l_0). We explain how the bridge vertices increase the matching along an augmenting path in Section 3.9.

If r_0 is unable to find an alternating path to an exposed vertex, then there is no augmenting path containing r_0. Thus r_0 sets erased (r_0) := **true** and sends an ERASED message to l_0. When l_0 receives the ERASED message, l_0 deletes r_0 from bridges (l_0). Then l_0 selects some other vertex r_1 from bridges (l_0) and sends a GO (l_0) message to r_1. Vertex l_0 continues selecting buddies from bridges (l_0) until either some buddy is successful or all of them fail. If some buddy of l_0 is successful, then there is an augmenting path. If all buddies of l_0 fail, then l_0 sets erased (l_0) := **true** and sends a NOTAUGMENTED message to stparent (l_0).

We have described the process of finding an augmenting path for l_0. Now we describe the process for a general vertex x.

If x receives a STARTAUGMENT message, then x forwards the STARTAUGMENT message to its leftmost ST-child, if one exists. If x receives an AUGMENTED or NOTAUGMENTED message from a ST-child of x, then x sends a STARTAUGMENT message to x's next ST-child from the left, if one exists. After x has sent a STARTAUGMENT message and received an

AUGMENTED or NOTAUGMENTED reply from each of x's ST-children, if x is not erased and x found bridges, then x tries to find an alternating path to an exposed vertex. If x is able to find one, then x sends a GO (x) message to the buddies of x in bridges (x) one at a time until either some buddy returns a SUCCESS message or all of them return ERASED messages.

If x receives a SUCCESS message, then x and x's buddy increase the matching along the augmenting path. If x was unable to find an augmenting path, then x sets erased (x) := **true**.

Then x sends an AUGMENTED or NOTAUGMENTED message to its ST-parent as follows. If either x or some ST-descendant of x found an augmenting path, then x sends an AUGMENTED message. Otherwise x sends a NOTAUGMENTED message. Thus if x found an augmenting path or some ST-child of x returned an AUGMENTED message, then x sends an AUGMENTED message to stparent (x). Otherwise, x sends a NOTAUGMENTED message to stparent (x).

Observe that during each phase x can belong to at most one augmenting path. Thus, once the algorithm finds an augmenting path containing x, x is erased and the bridges owned by x need not be considered.

The sending of the AUGMENTED and NOTAUGMENTED messages forms a convergecast that synchronizes the end of the search for augmenting paths containing the bridges found at the current search level. The convergecast also informs the leader whether there was an augmentation. If L receives an AUGMENTED message, then an augmentation occurred. Thus L increments the phase $p := p + 1$ and begins a new phase by broadcasting a STARTPHASE (p) message. If L receives only NOTAUGMENTED messages, then no augmentations occurred. In that case L increments the search level $i := i + 1$ and continues the search for bridges at the next level by broadcasting a STARTBFS (i) message.

We point out two modifications that could be made to reduce the number of messages. These modifications, however, do not reduce the overall message complexity of the algorithm.

One way to reduce the number of messages is to send STARTAUGMENT messages only to those vertices that sent BRIDGES messages. Vertices that sent NOBRIDGES messages and their ST-descendants do not need to be sent STARTAUGMENT messages since they did not find any bridges.

We also note that L does not need to initiate a search for an alternating path, i.e., L searches for an alternating path only if L receives a GO message. If there was an augmenting path containing l, then L is erased. If L is not erased, then there is no augmenting path containing L. This is because each buddy of L must have been either contained in an augmenting or unable to find an alternating path to an exposed vertex, and thus erased. Otherwise some buddy of L would have sent a GO message to L.

3.7 Alternating Depth First Search

In the sequential algorithm of Micali and Vazirani, the bridge vertices l_0 and r_0 of a bridge (l_0, r_0) perform depth first search concurrently to find alternating paths to exposed vertices. The concurrent depth first search proceeds in lock step, where the depth first search for l_0 proceeds to the next level only if its level is greater than or equal to the level of the depth first search for r_0.

A straightforward implementation of the concurrent depth first search in the distributed algorithm would be to have l_0 and r_0 synchronize across the bridge after each step of the depth first search. The synchronization after each step, however, is very inefficient because it introduces a large number of messages.

We reduce the number of messages by using our implementation which we call *alternating depth first search*. The difference between alternating depth first search and concurrent depth first search is that in the alternating depth first search, the algorithm first finds an alternating path from one bridge vertex, say l_0, to an exposed vertex and then tries to find a disjoint alternating path from r_0 to an exposed vertex.

We define a *complete alternating path* to be an alternating path from a bridge vertex to an exposed vertex. We define *extending an alternating path* to be the process of increasing the length of an alternating path. An alternating path may be extended until it is complete.

We now describe how the bridge vertices l_0 and r_0 find an augmenting path containing the bridge (l_0, r_0) using alternating depth first search. Vertex l_0 begins the depth first search by choosing a predecessor x_n from predecessors (l_0). Vertex l_0 sets dfschild $(l_0) := x_n$ and sends a DFS (l_0, r_0) message to x_n. A vertex receiving a DFS (l_0, r_0) message knows the depth first search is for l_0 and r_0 is l_0's buddy.

When x_n receives the DFS (l_0, r_0) message, x_n determines whether it can be added to the alternating path. First x_n checks erased (x_n) to determine if it is erased. If x_n is not erased, then it checks visited (x_n) to determine if it has been visited by some other bridge vertex. We consider the following cases:

Case 1: If x_n is erased, then x_n sends an ERASED message to l_0.

Case 2: If x_n is not erased and has not been visited, then x_n can be added to the alternating path. To add x_n to the alternating path, x_n sets visited $(x_n) := l_0$, the bridge vertex initiating the depth first search, and dfsparent $(x_n) := l_0$. Then x_n tries to extend the alternating path by sending a DFS (l_0, r_0) message to a predecessor x_{n-1}.

This process is repeated $n - 1$ more times adding the vertices $x_{n-1}, x_{n-2}, \cdots, x_1$ to the alternating path until the depth first search reaches an exposed vertex x_0. After x_0 is added to the alternating path, x_0 knows that the alternating path is complete. Thus x_0 sets altpath (x_0) := **true** since x_0 is a vertex of a complete alternating path and sends a SUCCESS message to dfsparent (x_0), x_1. The SUCCESS message signals that the depth first search has found a complete alternating path.

When vertex x_1 receives the SUCCESS message, x_1 knows that it is a vertex of a complete alternating path. Thus x_1 sets altpath (x_1) := **true** and sends a SUCCESS message to dfsparent (x_1). This process is repeated until x_n sends a SUCCESS message to l_0.

When l_0 receives the SUCCESS message, l_0 sets altpath (l_0) := **true**. Vertex l_0 then signals buddy r_0 to proceed with depth first search by sending a GO (l_0) message to r_0. Vertex r_0 then performs depth first search and tries to find a complete alternating path. If during the depth first search a vertex x_i with visited $(x_i) = l_0$ receives a DFS (r_0, l_0) message, then x_i knows that it is a common vertex found by the depth first searches of l_0 and r_0. We discuss common vertices in Section 3.8.

Case 3: If x_n is not erased but was visited by some other bridge vertex, then x_n is a vertex of a previously discovered complete alternating path and can be added to the current alternating path. A previously discovered complete alternating path exists if when the algorithm was searching for an augmenting path containing another bridge, the algorithm found only one complete alternating path from the bridge. If both bridge vertices were able to find disjoint complete alternating paths, then there would be an augmenting path and x_n would be erased. If there was no alternating path from x_n to an exposed vertex, then again x_n would have been erased. Thus x_n must be a vertex of a previously discovered complete alternating path. However, the alternating path containing x_n may no longer be complete because another augmenting path may have subsequently passed through it. Thus some of the vertices may be erased.

To add x_n to the alternating path, x_n sets visited $(x_n) := l_0$ and dfsparent $(x_n) := l_0$. Vertex x_n sets altpath $(x_n) :=$ **false** because x_n does not know whether the alternating path to the exposed vertex is still complete. Then x_n sends a DFS (l_0, r_0) message to dfschild (x_n), vertex y_{n-1}, the child of x_n in the previous alternating path.

If the previous alternating path $x_n, y_{n-1}, \cdots, y_0$ is still complete, then the depth first search reaches the exposed vertex y_0 using the minimum number of messages. The intermediate vertices y_i set visited $(y_i) := l_0$ and altpath $(y_i) :=$ **false**. When y_0 receives the DFS (l_0, r_0) message, y_0 sets visited $(y_0) := l_0$, altpath $(y_0) :=$ **true**, and sends a SUCCESS message to dfsparent (y_0). Intermediate vertices y_i receiving a SUCCESS message set altpath $(y_i) :=$ **true** and forward a SUCCESS message to dfsparent (y_i). When x_n receives a SUCCESS message from y_{n-1}, x_n sets altpath (x_n) $:=$ **true** and sends a SUCCESS message to l_0.

If the search along the previously complete alternating path finds a vertex that is erased, then the depth first search sends a DFS (l_0, r_0) message to some other predecessor and the depth first search proceeds as if the previously discovered complete alternating path did not exist.

We have described the depth first search for x_0. We now digress briefly and describe the depth first search for a general vertex w_i. Suppose w_i receives a DFS (a, b) message from w_{i+1}.

If w_i is erased, then w_i sends an ERASED message to w_{i+1}. If w_i can be added to the alternating path, w_i sets dfsparent (w_i) := w_{i+1} and visited (w_i) := a.

If w_i is exposed, then w_i sets altpath (w_i) := **true** and sends a SUCCESS message to dfsparent (w_i). Otherwise, w_i tries to extend the alternating path. If w_i is a vertex of a previously discovered complete alternating path, then w_i sends a DFS (a, b) message to dfschild (w_i). If w_i has not been visited, then w_i selects a predecessor w_{i-1} from predecessors (w_i), sets dfschild (w_i) := w_{i-1}, and sends a DFS (a, b) message to w_{i-1}.

If w_i receives an ERASED message from dfschild (w_i), then w_i deletes dfschild (w_i) from predecessors (w_i) and sends a DFS (a, b) message to some other predecessor of w_i. If w_i receives a SUCCESS message from some predecessor of w_i, then w_i sets altpath (w_i) := **true** and sends a SUCCESS message to dfsparent (w_i). If all predecessors of w_i return ERASED messages, then w_i sets erased (w_i) := **true** and sends an ERASED message to dfsparent (w_i).

We now return to the alternating depth first search with the bridge vertices l_0 and r_0. Vertex l_0 sends DFS (l_0, r_0) messages to its predecessors one a time until some predecessor of l_0 returns a SUCCESS message or all predecessors of l_0 return ERASED messages.

If all of l_0's predecessors return ERASED messages, then there is no augmenting path containing l_0. Thus l_0 sets erased (l_0) := **true** and sends a NOTAUGMENTED message to stparent (l_0).

If vertex l_0 receives a SUCCESS message, then there is an alternating path from l_0 to an exposed vertex. Vertex l_0 then sends a GO (l_0) message to r_0.

If r_0 is erased, then r_0 sends an ERASED message to l_0. Otherwise, r_0 tries to find an alternating path to an exposed vertex. The depth first search for r_0 is the same as for l_0.

If r_0 finds a complete alternating path, then r_0 sends a SUCCESS message to l_0. AFter l_0 and r_0 increase the matching along the augmenting path, l_0 sends an AUGMENTED message to stparent (l_0).

If r_0 is unable to find an alternating path to an exposed vertex, then there is no augmenting path containing r_0. Vertex r_0 sets erased (r_0) := **true** and sends an ERASED message to l_0.

If l_0 receives an ERASED message from r_0, then l_0 deletes r_0 from bridges (l_0) and selects some other buddy r from bridges (l_0). If all buddies of l_0 return ERASED messages, then there is no augmenting path containing l_0. In that case, l_0 sets erased (l_0) := **true** and sends a NOTAUGMENTED message to stparent (l_0).

We give a short example for finding an augmenting path. In Figure 6, suppose we are searching for an augmenting path containing the bridge (A, D) and A is ready to begin alternating depth first search. A sends a DFS (A, D) message to B, and then B sends a DFS (A, D) message to C. Since C is exposed, C sends a SUCCESS message to B, and then B sends a SUCCESS message to A. Then A

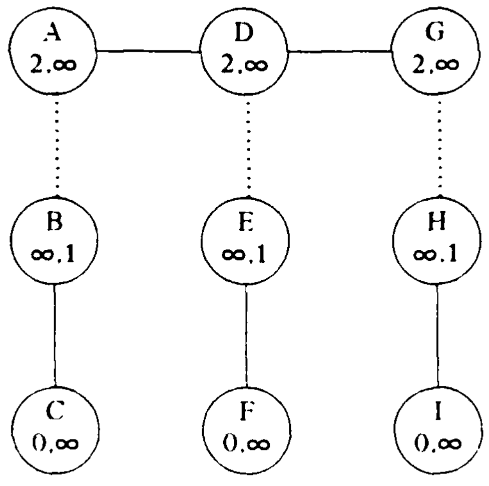


Figure 6. Finding an Augmenting Path

sends a GO (A) message to D. D sends a DFS (D, A) message to E and then E sends a DFS (D, A) message to F. Since F is exposed, F sends a SUCCESS message to E. When D receives a SUCCESS message from E, D sends a SUCCESS message to A. Thus A and D find an augmenting path. As A and D increase the matching along the augmenting path (C, B, A, D, E, F), vertices A, B, C, D, E, and F are erased. Thus when G tries to find an augmenting path for the bridge (G, D) and sends a GO (G) message to D, D sends an ERASED message to G.

3.8 Common Vertices and Backtracking

A *common vertex* is a vertex discovered by both vertices of a bridge during alternating depth first search. Suppose that we are trying to find an augmenting path containing the bridge (l, r) and that l finds an alternating path to an exposed vertex. If the depth first search for r encounters a vertex c with visited $(c) = l$, then c is a common vertex of l and r. The *deepest common vertex* (DCV) is the common vertex with the smallest level found so far by both l and r during alternating depth first search.

We define *backtracking* to be the process of the depth first search trying to find a different alternating path to an exposed vertex. To avoid redundant backtracking over edges that have already been searched, each vertex v maintains a variable barrier (v) used to keep track of how far the search has progressed. Backtracking is not allowed to back up beyond a vertex that has already been searched.

We now describe the alternating depth first search with common vertices. Figure 7 illustrates the relationship between the vertices in our discussion of the alternating depth first search. The dashed lines represent omitted vertices.

After l finds a complete alternating path, l sets barrier $(l) := l$ since the depth first search for l should not backtrack beyond l. Vertex l sets barrier (l) only if l finds a complete alternating path. Otherwise, l would be erased. Then l sends a GO (l) message to r. When r receives the GO (l) message, r sets barrier $(r) := r$ and begins depth first search.

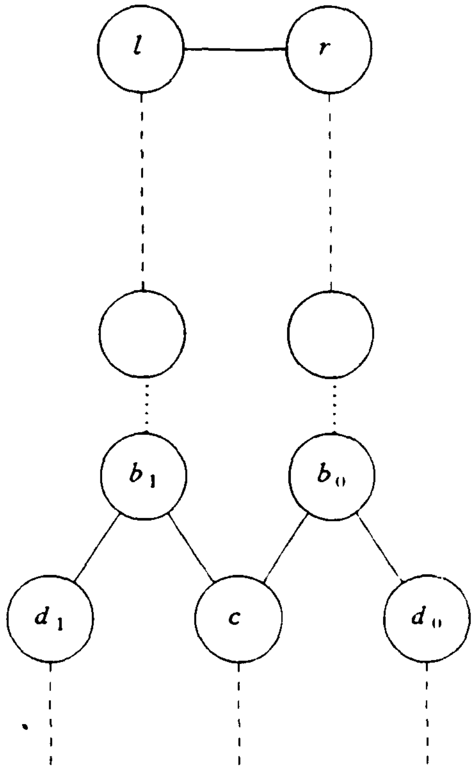


Figure 7. Illustration for Alternating Depth First Search

A vertex c realizes it is a common vertex of l and r if c receives a DFS (r , l) message and visited $(c) = l$. Since l was the first bridge vertex to find c, the depth first search for r backtracks first. Suppose the DFS (r , l) message sent to c came from vertex b_0. Then c causes the depth first search for r to backtrack by sending a BACKTRACK (c , r) message to b_0.

When b_0 receives the BACKTRACK (c , r) message from c, b_0 knows that c is the next vertex on the path from b_0 to the DCV and sets dcvchild $(b_0) := c$. Vertex b_0 tries to find another alternating path by sending a DFS (r , l) message to some predecessor d_0 of b_0, if one exists.

If there is an alternating path to an exposed vertex through d_0 disjoint from the one found by l, then there is an augmenting path.

If d_0 is unable to find a complete alternating path, then d_0 is erased and sends an ERASED message to b_0. If all other predecessors of b_0 return ERASED messages, then b_0 sends a BACKTRACK (c, r) message to dfsparent (b_0). Vertex b_0 is not erased since it has an alternating path to an exposed vertex through the DCV.

This process continues up the alternating path of the depth first search for r until either a vertex of the alternating path finds a complete alternating path and thus an augmenting path, or r receives a BACKTRACK (c, r) message.

If r receives a BACKTRACK (c, r) message and is able to find a complete alternating path through another predecessor, then there is an augmenting path. Otherwise, since barrier $(r) = r$, r does not backtrack any further. Vertex r must then claim the DCV and force l to backtrack. In addition, by claiming the DCV, r indirectly claims the alternating path leading from the DCV to the exposed vertex via the dfschild variables.

To claim the DCV, r sends a TAKINGDCV (r) message to l to inform l that r is claiming the DCV. When l receives the TAKINGDCV (r) message, l knows that it can receive BACKTRACK and SUCCESS messages from its predecessors as a result of backtracking. Then r sends a CLAIMDCV (r, c) message which is forwarded along the alternating path via the dcvchild variables until it reaches c. When c receives the CLAIMDCV (r, c) message from b_0, c sets visited $(c) := r$ and barrier $(c) := r$ since the depth first search by r should not backtrack beyond c. Vertex c sends a BACKTRACK (c, l) message to dfsparent (c), vertex b_1, and then sets dfsparent $(c) := b_0$.

After receiving the BACKTRACK (c, l) message, b_1 sends a DFS (l, r) message to some predecessor d_1, if one exists. The process for d_1 is the same as for d_0. The backtracking process repeats up the alternating path of the depth first search for l until either some vertex of the alternating path finds a complete alternating path, or l receives a BACKTRACK (c, l) message.

If some vertex of the alternating path is able to reach an exposed vertex, then l receives a SUCCESS message. After l receives the SUCCESS message, l knows that backtracking was successful and that there is an augmenting path.

If none of l's other predecessors is able to find another alternating path to an exposed vertex, then l must claim the DCV and force the depth first search for r to backtrack. Thus l sends a CLAIMDCV (l, c) message to dcvchild (l).

The intermediate vertices forward the CLAIMDCV (l, c) message until it reaches c. When c receives the CLAIMDCV (l, c) message, c realizes that l is forcing r to backtrack. But since barrier (c) := r, the depth first search for r cannot backtrack beyond c. Thus c knows that there is no augmenting path containing the bridge (l, r). In fact, the alternating depth first search from l and r has discovered a blossom with base c. We discuss blossoms in Section 3.10.

While backtracking, it is possible that the depth first search for l may find another common vertex c_1 at a level lower than the level of the DCV claimed by r. This situation is recognized by c_1 if c_1 receives a DFS (l, r) message and visited (c_1) = l. This is because the only way l could visit c_1 again is if the depth first search of l was forced to backtrack and found another alternating path to c_1. Thus c_1 forces the depth first search for l to backtrack.

If l is unable to find a different alternating path to an exposed vertex, then l sends a CLAIMDCV (l, c_1) message to claim c_1. Vertex l does not need to send a TAKINGDCV message to r since r already knows there is a common vertex. When c_1 receives the CLAIMDCV (l, c_1) message, c_1 sets barrier (c_1) := l and forces the depth first search for r to backtrack.

If the depth first search for r cannot find another complete alternating path, then the depth first search for r will backtrack to c. Since barrier (c) = r, the depth first search for r does not backtrack further. Vertex c must claim c_1. Thus c sends a CLAIMDCV (r, c_1) message which is forwarded to c_1. When c_1 receives the CLAIMDCV (r, c_1) message and finds barrier (c_1) = l, c_1 knows there is a blossom with base c_1. The original exposed vertex found by l may be alternately claimed by l and r several times.

We now give an example to demonstrate alternating depth first search with common vertices. Suppose we have the graph shown in Figure 8 and the algorithm is searching for an augmenting path from the bridge (A, B).

Suppose A goes first and finds the complete alternating path (A, D, F, H, J, K, M, P). Thus vertices A, D, F, H, J, K, M, and P all have their visited variables set to A. Vertex A sets barrier (A) := A and sends a GO (A) message to B signaling B to proceed.

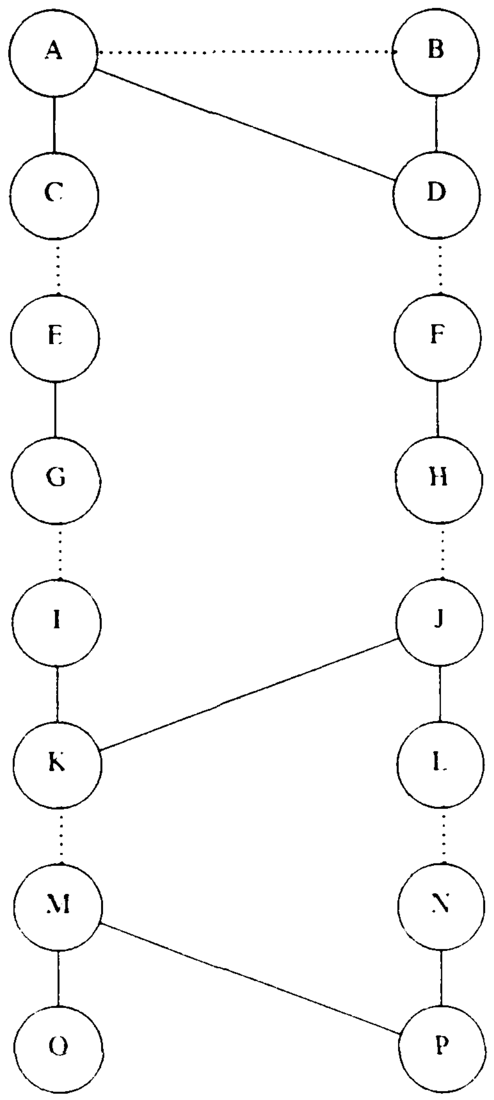


Figure 8. Common Vertices and Backtracking

When B receives the GO message. B sets barrier (B) := B. B sends a DFS (B. A) message to D. D discovers that it is a common vertex of A and B since D has already been visited by A. Thus D sends a BACKTRACK (B. D) message to B. When B receives the BACKTRACK (B. D) message from D. B sets dcvchild (B) := D. since D is the next vertex along the path leading to the DCV. Since B has no other predecessors. B must claim the DCV D. Thus B sends a TAKINGDCV (B) message to A notifying A that B is claiming the DCV. Then B sends a CLAIMDCV (B. D) message to D to claim the DCV.

When D receives the CLAIMDCV (B. D) message. D sets barrier (D) := B. and visited (D) := B. After D sends a BACKTRACK (A. D) message to A. D sets dfsparent (D) := B.

When A receives the BACKTRACK (A. D) message. A sets dcvchild (A) := D and sends a DFS (A. B) message to predecessor C. The depth first search for A progresses to vertices E. G. and I. When K receives a DFS (A. B) message from I. K knows it is a common vertex since K has already been visited by A.

K sends a BACKTRACK (A. K) message to I. I sets dcvchild (I) := K and sends a BACKTRACK (A. K) message to G since I does not have any other predecessors. Eventually A receives a BACKTRACK (A. K) message from C. Since A has no other predecessors. A sends a CLAIMDCV (A. K) message to K which is forwarded by the intermediate vertices using their dcvchild variables until it reaches K. A does not send a TAKINGDCV message to B since B already knows there is a common vertex.

When K receives the CLAIMDCV (A. K) message. K sets visited (K) := A and barrier (K) := A. After K sends a BACKTRACK (B. K) message to J. K sets dfsparent (K) := I.

When J receives the BACKTRACK (B. K) message. J sets dcvchild (J) := K. and sends a DFS (B. A) message to predecessor L. The depth first search for B progresses to N. When P receives a DFS (B. A) message from N. P knows that it is a common vertex because it has already been visited by A. Thus P sends a BACKTRACK (B. P) message message to N. The depth first search backtracks until it reaches D. Since barrier (D) = B. further backtracking for B is prohibited. Since there are no other predecessors of D. D sends a CLAIMDCV (B. P) message to P. When P receives the CLAIMDCV (B. P) message. P sets visited (P) := B and barrier (P) := B. After P sends a BACKTRACK (A. P) message to M. P sets dfsparent (P) := N.

When M receives the BACKTRACK (A. P) message from P. M sets dcvchild (M) := P and sends a DFS (A. B) message to predecessor O. Since O is exposed. O sends a SUCCESS message to M which is eventually forwarded to A. When A receives the SUCCESS message. A knows there is an augmenting path.

If vertex O did not exist. then K would have received a BACKTRACK (A. P) message. Since K has barrier (K) = A. K sends a CLAIMDCV (A. P) message to M. which is forwarded to P. When P receives the CLAIMDCV (A. P) message and finds barrier (P) = B. then P knows there is a blossom with base P.

3.9 Increasing the Matching

If an augmenting path is discovered. the algorithm obtains a new matching of greater cardinality by reversing the matching of the edges of the augmenting path. Since the number of free edges in an augmenting path is one more than the number of matched edges. the cardinality of the new matching increases by 1.

Suppose the bridge vertices of a bridge (l, r) have found disjoint alternating paths to exposed vertices. We describe how l and r reverse the matching of the edges of the augmenting path.

After l receives a SUCCESS message from r. l sends a STARTINVERT message to dfschild (l) that is forwarded along the alternating path until it reaches the exposed vertex z_l. Intermediate vertices x_i forward the STARTINVERT message to dfschild (x_i). When z_l receives the STARTINVERT message. z_l knows that it is a vertex of an augmenting path. Thus z_l sets erased (z_l) := **true**. Then z_l must reverse the matching of its edge in the augmenting path. Since z_l is exposed. the edge between z_l and dfsparent (z_l) is free. To reverse the matching of the edge. z_l sets mate (z_l) := dfsparent (z_l). Then z_l sends an ENDINVERT message to dfsparent (z_l).

An intermediate vertex x_i receiving an ENDINVERT message from dfschild (x_i) sets erased (x_i) := **true**. To reverse the matching of its edges in the augmenting path, if x_i is currently matched with its DFS-child, then x_i matches itself with its DFS-parent, and vice versa. Thus if mate (x_i) = dfschild (x_i), then x_i sets mate (x_i) := dfsparent (x_i). If mate (x_i) = dfsparent (x_i), then x_i sets mate (x_i) := dfschild (x_i). Then x_i sends an ENDINVERT message to dfsparent (x_i).

When l receives an ENDINVERT message, l sets erased (l) := **true**. Then l reverses the matching of its edges. If mate $(l) = r$, then l sets mate (l) := dfschild (l), and if mate (l) = dfschild (l), then l sets mate $(l) := r$.

The process of increasing the matching for r along its complete alternating path is the same. After r receives an ENDINVERT message and sets erased (r) and mate (r), r sends an ENDINVERT message to l. After l has set erased (l) and mate (l), l sends an AUGMENTED message to stparent (l).

3.10 Blossoms

Now we consider general graphs with blossoms. We start by describing blossoms. We use the description of Peterson (1985). A blossom exists if there is a bridge (s, t) and vertices a such that a is an ancestor of both s and t, and no ancestors of s and t other than a have level equal to level (a). Among the set of vertices a, let b be the vertex whose level is maximum. Then the blossom B is the set of vertices d such that:

(1) d does not belong to any other blossom when B is formed,

(2) b is an ancestor of d, and

(3) either $d = s$ or $d = t$ or d is an ancestor of s or of t.

Vertex b is the *base* of blossom B. Figure 9a is a graph with the blossom (C, D, E, F, G) with base C. An *embedded blossom* is a blossom whose base belongs to another blossom. A blossom may be embedded in more than one blossom.

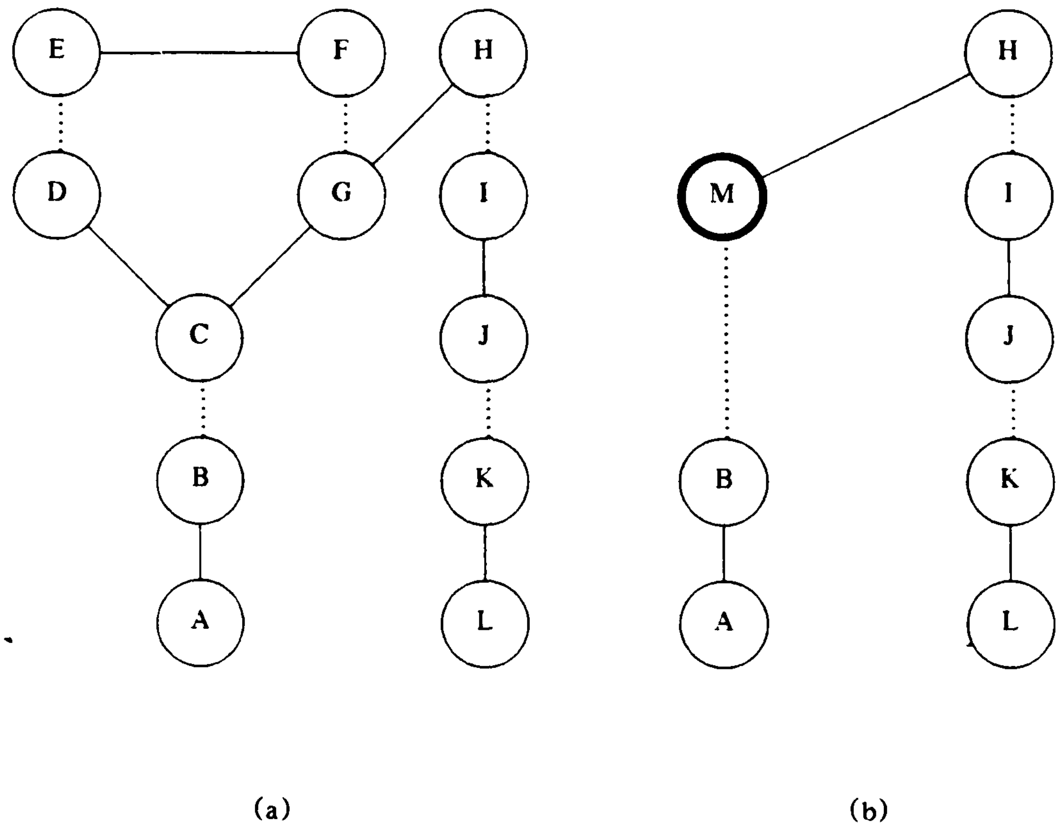


Figure 9. Shrinking a Blossom

Next we describe a method of handling blossoms used by several sequential algorithms. Edmonds (1965) presented the idea of *shrinking* blossoms by replacing each blossom with a single "supervertex". Figure 9b shows the result of replacing the blossom (C, D, E, F, G) in Figure 9a by the supervertex M. Edmonds proved the following theorem:

Theorem 3.1: Let G be a graph with a blossom. Let G' be the graph obtained from G by shrinking the blossom. Then there is an augmenting path in G if and only if there is an augmenting path in G'.

Micali and Vazirani used this idea of shrinking blossoms to attain the $O(|E|)$ time for each phase of their sequential algorithm. In their algorithm, if the depth first search for an exposed vertex encounters a vertex belonging to a blossom, the search "jumps" to the base of the blossom and continues the search. By making this jump, their algorithm avoids repeated traversals of the edges of the blossom. If the search finds an augmenting path, the blossom is *opened* to obtain the complete augmenting path.

In a distributed system the notion of "jumping" does not apply. Since each vertex can only communicate with its neighbors, each "jump" would require the vertices of the blossom to send a message along the blossom until it reached the base. Thus shrinking blossoms would not significantly reduce the number of messages.

We now describe the execution of our algorithm in graphs with blossoms. Since the steps of the algorithm have been described in detail in the previous sections, we give a higher level description.

The main difference between searching for augmenting paths in graphs with blossoms and graphs without blossoms is the presence of anomalies. In a graph where there are blossoms, the edge between a vertex g and an anomaly f is a bridge. Thus the presence of anomalies may lead to the discovery of additional augmenting paths. In a graph without blossoms, the edge between g and f is not a bridge.

We return to the execution of the algorithm in Section 3.8. The alternating depth first search for an augmenting path containing bridge (l, r) discovers a common vertex c. If c receives a CLAIMDCV (l, c) message from b_1 and barrier $(c) = r$, then c knows there is a blossom with base c. To notify l and r that there is a blossom, c sends a BLOS (l, c) message to b_1 which is forwarded to l and a BLOS (r, c) message which is forwarded to r. Vertices l and r realize there is a blossom with base c when they receive the BLOS messages.

Let B be the blossom with base c and let t be the tenacity (l, r). Then for each vertex b in B except the base c, the algorithm sets blossom $(b) := c$ and the otherlevel of b to t - level (b).

We call this process *labeling* the blossom.

We point out a special characteristic of vertices belonging to a blossom. Each vertex b belonging to a blossom has two alternating paths to an exposed vertex. One path is the "direct" path and the other goes "around" the blossom. One path has even length and the other odd length. One path begins with a free edge and the other begins with a matched edge. Thus each vertex b of the blossom has an alternating path to an exposed vertex through both dfsparent (v) and dfschild (v), except the bridge vertices which have the their buddy instead of dfsparent (v). Thus, for each vertex b in the blossom, the algorithm sets altpath (b) := **true**.

To label the blossom B, l and r compute t. Note that since level (l) = level (r), l and r can compute t independently. We describe how l labels the vertices b between l and c. Vertex l sets blossom (l) := c and altpath (l) := **true**. Then l sends a BLOSSOM (l, c, t) message to dcvchild (l), a vertex b. Vertices b receiving a BLOSSOM (l, c, t) message that do not already belong to a blossom set blossom (b) := c, the otherlevel of b to t - level (b), and altpath (b) := **true**. Then b forwards the BLOSSOM (l, c, t) message to dcvchild (b).

The BLOSSOM (l, c, t) message is forwarded until it reaches c. If a vertex d receiving a BLOSSOM (l, c, t) belongs to an embedded blossom, then d just forwards the BLOSSOM message to dcvchid (d).

When c receives the BLOSSOM (l, c, t) message from b_1, c sends a BLOSSOMREPLY (l) message to b_1. Vertex c already has altpath (c) = **true**. The BLOSSOMREPLY message is forwarded via dfsparent to l.

The process of labeling the blossom for r is the same. After r receives a BLOSSOMREPLY (r) message, r sends a LABELED message to l. After l has received a BLOSSOMREPLY (l) message and a LABELED message from r, l knows the blossom is labeled. Figure 10 shows a blossom that has been labeled.

Since a blossom has been discovered, there is no augmenting path containing bridge (l, r). Vertex l checks bridges (l) to determine if l found any other bridges. If not, then l sends a

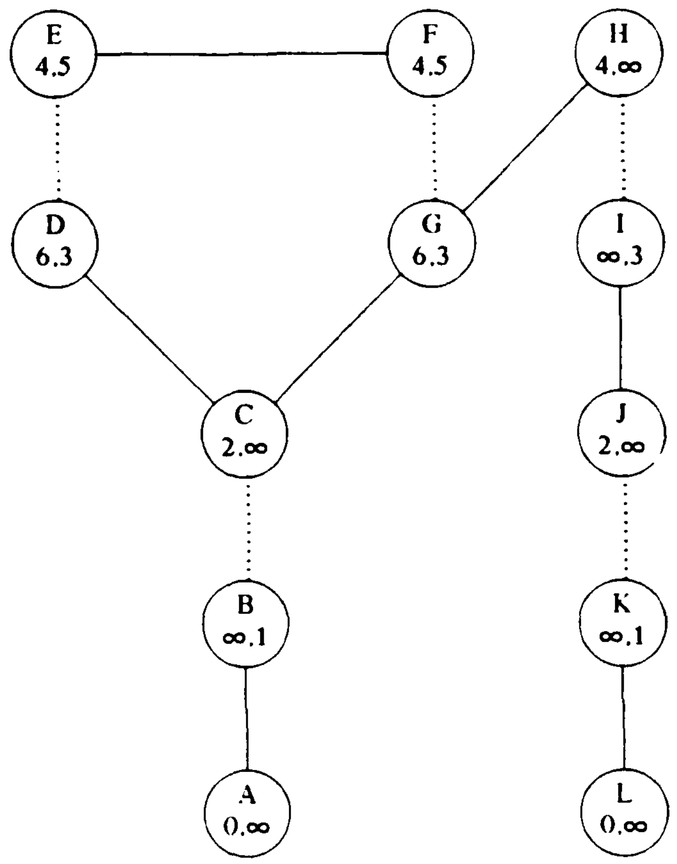


Figure 10. A Labeled Blossom

NOTAUGMENTED message to stparent (l). Note that l is not erased since l has a complete alternating path.

If l found another bridge with a vertex s, then l sends a GO (l) message to s. If s is able to find a disjoint alternating path to an exposed vertex, then there is an augmenting path. Then l and s increase the matching along the augmenting path, and l sends an AUGMENTED message to stparent (l).

If the depth first search for s finds a vertex b of the blossom B found by l and r, then since altpath (b) = **true**, the depth first search follows the complete alternating path to an exposed vertex. Thus s receives a SUCCESS message and s sends a SUCCESS message to l. To determine whether the alternating path found by s has any common vertices with the complete alternating path for l, vertex l searches its complete alternating path again. We accomplish this by modifying the DFS message to signify that it is re-searching a path. If the paths are disjoint, then there is an augmenting path. If there is a common vertex, then the alternating depth first search for l and s is the same as in the case with common vertices. If s and l find another blossom, then s and l label the blossom with the base. Note that it is possible that the blossom found by l and r has the same base as the blossom found by l and s. Then the vertices of both blossoms would have blossom () set to c.

If the depth first search to find an exposed vertex for some other bridge found at the current search level or at a higher search level finds a vertex belonging to a blossom, then the search proceeds in the same way as in the case of finding a previous complete alternating path.

When the algorithm has completed the search for augmenting paths at the current search level, if the leader received an AUGMENTED message, then the leader begins a new phase, and the discovery of blossoms at the current search level has no effect. If no augmenting paths were found, then the leader continues the breadth first search by incrementing the search level $i := i + 1$ and broadcasting a STARTBFS (i) message.

As the leader increases the search level, we want the breadth first search for bridges to wrap around blossoms. If the search discovers a bridge such that both vertices of the bridge belong to the same blossom B, then we ignore the bridge because a search for an augmenting path would lead to the rediscovery of B. Thus, we modify the BRIDGESEARCH message to tell whether a vertex x belongs to a blossom. If x belongs to a blossom, then the BRIDGESEARCH message includes the base of this blossom.

To allow the search for bridges to wrap around blossoms, we modify the breadth first search so that when the leader broadcasts a STARTBFS (i) message, vertices v may send BRIDGESEARCH messages if either evenlevel (v) = i or oddlevel (v) = i. Vertices v send BRIDGESEARCH messages to vertices a in anomalies(v) and those vertices u along alternating paths in activeneighbors(v) - predecessors(v) to which v has not sent BRIDGESEARCH messages previously.

If v has an anomaly a, then the edge (v, a) is a bridge, and v convergecasts a BRIDGES message. If v receives a BRIDGEREPLY (b, **bridge**) message from a vertex u, then the edge (v, b) is a bridge, and v convergecasts a BRIDGES message. Otherwise, v convergecasts a NOBRIDGES message. As the search level is increased, the breadth first search follows a path leading out of the blossom, if one exists.

If a bridge is discovered, the search for augmenting paths is basically the same as before. The only difference is that during the depth first search to find an alternating path to an exposed vertex, the vertices in a blossom need to consider two kinds of predecessors, those joined by a free edge and those joined by a matched edge. Thus, if a vertex b belonging to blossom is searching for an exposed vertex or backtracking, b needs to select a predecessor of the correct kind. But this is simple since b knows the matching of the edge on which the last DFS or BACKTRACK message arrived. Thus, if the last DFS message arrived on a matched edge, then b sends a DFS message to a predecessor joined by a free edge, and vice versa. If the last BACKTRACK message arrived on a free edge, then b selects another predecessor joined by a free edge since b tries to extend the alternating path. Note that each vertex has at most one matched predecessor.

The process of increasing the matching along the augmenting path is the same as before. The STARTINVERT and ENDINVERT messages are sent via the dfschild and dfsparent variables.

We give a brief example describing how the algorithm finds augmenting paths in graphs with blossoms. Figure 11 is a graph with an embedded blossom. The current matching is the matching shown.

The search for bridges begins from the exposed vertices A and T. When the search level is 4, vertex O discovers anomaly P since P sends a BRIDGESEARCH message to O. Thus O knows the edge (O, P) is a bridge if O belongs to a blossom. However, at this time, O does not know whether

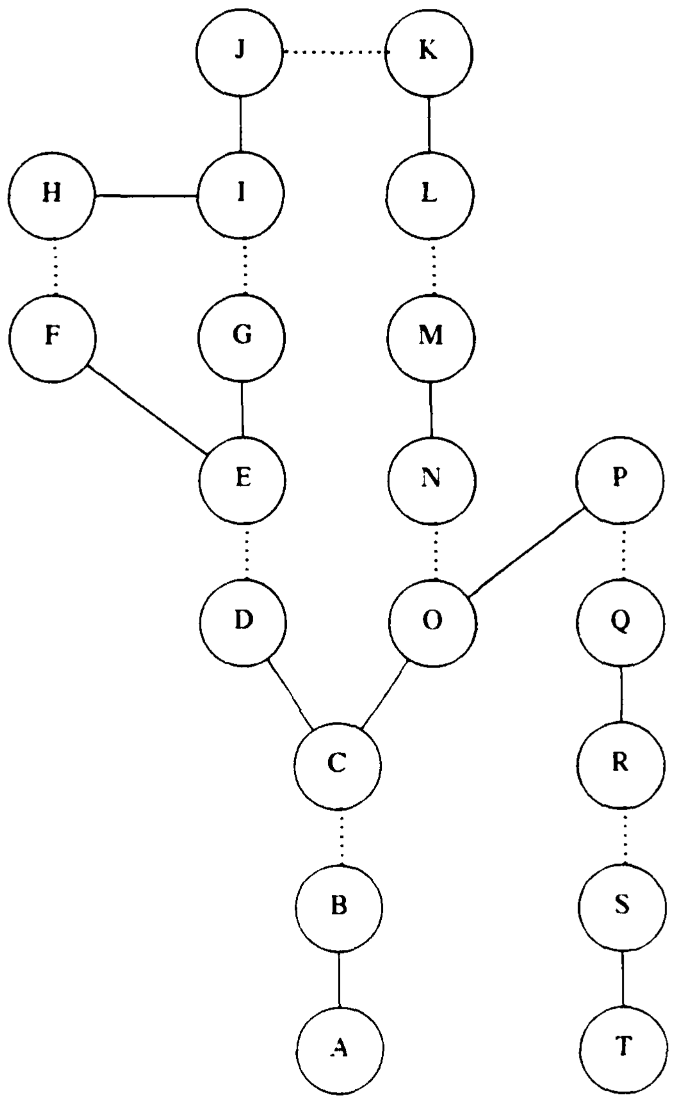


Figure 11. An Embedded Blossom

it belongs to a blossom. Since there is no augmenting path, the search for bridges continues.

When the search level reaches 6, H and I discover the bridge (H, I). The search for an augmenting path from H and I leads to the discovery of the blossom (E, F, G, H, I). Thus F, G, H, and I are labeled with base E. Since there is no augmenting path, the breadth first search continues. At search level 7, the bridge (J, K) is discovered. The search for an augmenting path leads to the discovery of the blossom (C, D, E, G, I, J, K, L, M, N, O). The vertices of this blossom are labeled with base C, except G and I which were already labeled with base E. Note that blossom (E) = C.

Since there again is no augmenting path, the breadth first search continues and wraps around the blossom. When the search level reaches 12, vertex O convergecasts a BRIDGES message since O has the anomaly P. The search for an augmenting path begins with vertex O. By performing depth first search, O finds the complete alternating path (O, N, M, L, K, J, I, G, E, D, C, B, A). Then P finds the complete alternating path (P, Q, R, S, T). Then O and P proceed to increase the matching along the augmenting path (A, B, C, D, E, G, I, J, K, L, M, N, O, P, Q, R, S, T).

CHAPTER 4

ANALYSIS

4.1 Correctness

We show informally that the algorithm is correct. At the start of a phase, if the current matching M is maximum, then by Berge's Theorem there is no augmenting path. Thus the breadth first search for bridges in the phase reaches a level i such that no vertex v has either evenlevel $(v) = i$ or oddlevel $(v) = i$, and the algorithm halts.

If the current matching M is not maximum, then by Berge's Theorem there is an augmenting path P with respect to M. Since the search for bridges continues until it reaches a level i where an augmenting path is found, the algorithm finds an augmenting path and increases the matching. Since the search for bridges at level i finds all the bridges at level i, and the algorithm tries to find augmenting paths containing every bridge, the algorithm finds a maximal set of equal length augmenting paths. Since the algorithm begins a new phase if it finds an augmenting path at the current search level, the augmenting paths it finds during each phase must be a maximal set of minimum length augmenting paths.

4.2 Message Complexity

We determine the message complexity of our distributed matching algorithm.

Theorem 4.1: The message complexity of our distributed algorithm in the worst case is $O(|V|^{5/2})$ messages.

We prove Theorem 4.1 by adding up the total number of messages in the worst case. We first determine the number of messages required during preprocessing to construct a spanning tree and to select the root to be the leader. If we use the distributed algorithm of Gallagher, Humblet, and Spira (1983) for minimum weight spanning trees, then the number of messages is $O(|V|\log|V| + |E|)$. If we use the distributed algorithm of Awerbuch (1985b) for depth first search spanning trees, then the number of messages is $O(|E|)$.

Now we determine the number of messages required to compute a maximum matching using our algorithm given that the spanning tree has been constructed. During each phase, the algorithm finds a maximal set of minimum length augmenting paths. Hopcroft and Karp (1973) proved that no more than $O(|V|^{1/2})$ such phases are needed to find a maximum matching. During each phase, the number of search levels is no more than $|V|$. Thus the total number of search levels required by the algorithm is $O(|V|^{3/2})$.

At the start of each phase, the leader broadcasts a STARTPHASE message. Since vertices send the STARTPHASE messages along the spanning tree, there are $|V|$ STARTPHASE messages per phase. Each vertex convergecasts a READY message to end the initialization of a phase. Since the READY messages are also sent along the spanning tree, the number of READY messages per phase is also $|V|$. Thus the number of STARTPHASE and READY messages for the algorithm is $O(|V|^{3/2})$.

To conduct the search for bridges, the leader broadcasts a STARTBFS message for each search level. Since the STARTBFS messages are sent along the spanning tree, the number of STARTBFS messages is $|V|$ per search level. Thus the number of STARTBFS messages for the algorithm is $O(|V|^{5/2})$.

During a phase, each vertex sends at most one BRIDGESEARCH message over an edge. Since at most two BRIDGESEARCH messages are sent over each edge, the number of BRIDGESEARCH messages is $O(|E|)$ per phase. Thus the number of BRIDGESEARCH messages for the algorithm is $O(|V|^{1/2}|E|)$. Since there is one BRIDGEREPLY message for each BRIDGESEARCH message, the number of BRIDGEREPLY messages for the algorithm is also $O(|V|^{1/2}|E|)$.

To report to the leader whether any bridges were discovered at each search level, each vertex sends either a BRIDGES or NOBRIDGES message to its ST-parent. Since each vertex sends only one of these messages at each search level, the number of BRIDGES and NOBRIDGES messages for each search level is $|V|$. Thus the number of BRIDGES and NOBRIDGES messages for the algorithm is

O($|V|^{5/2}$).

At each search level, each vertex sends at most one STARTAUGMENT message. Thus the number of STARTAUGMENT messages for the algorithm is O($|V|^{5/2}$). Since there is one AUGMENTED or NOTAUGMENTED message for each STARTAUGMENT message, the number of AUGMENTED and NOTAUGMENTED messages for the algorithm is also O($|V|^{5/2}$).

To compute the number of DFS messages, we consider two cases in which the algorithm sends DFS messages.

In the first case, we consider alternating depth first search and backtracking where there are no previously complete alternating paths. During each phase, the algorithm sends at most one DFS message over an edge in each direction. If a vertex v receives a DFS message from a vertex w and there is no alternating path from v to an exposed vertex, then v sends an ERASED message to w and w sends no more DFS messages to v. If v finds an alternating path, then by our assumption, v must belong to an augmenting path. Thus, after the algorithm increases the matching, v and w are erased and w does not send another DFS message to v during the phase. Since there are $|E|$ edges, the number of DFS during each phase for this case is O($|E|$).

In the second case, we consider the presence of previously discovered complete alternating paths. If alternating depth first search finds an unerased vertex v with altpath (v) = **true**, then the depth first search retraces edges of the alternating path. We count these additional DFS messages. If the alternating path is still complete, then the number of additional DFS messages is exactly the length of the path. If the alternating path is no longer complete, then the DFS messages sent in order to find another alternating path are counted in the first case. Thus the only additional DFS messages are those sent along previously complete alternating paths. Note that during each phase, each vertex belonging to a previously discovered complete alternating path is visited by no more than $|V|$ different vertices. Since the combined lengths of the complete alternating paths during a phase is at most $|V|$ and each vertex of such a path can be visited no more than $|V|$ times, the number of DFS messages in the second case is at most $|V|^2$ per phase. Thus the total number of

DFS messages for the algorithm is $O(|V|^{1/2}|E|) + O(|V|^{5/2}) = O(|V|^{5/2})$.

A vertex sends a SUCCESS message when it discovers that it is a vertex belonging to a complete alternating path. Similar to the second case with the DFS messages, a vertex v of a previously complete alternating paths may send more than one SUCCESS message if the alternating depth first search finds another complete alternating path containing v. Thus the number of SUCCESS messages for the algorithm is $O(|V|^{5/2})$.

During the alternating depth first search, if a vertex v receiving a DFS or BACKTRACK message from a vertex w is unable to find an alternating path to an exposed vertex, v erases itself and sends an ERASED message to w. The ERASED message has the effect of removing the edge (v, w) for the remainder of the phase when performing alternating depth first search. Since there are $|E|$ edges, the number of ERASED messages during each phase is at most $|E|$. The number of ERASED messages for the algorithm is $O(|V|^{1/2}|E|)$.

During a phase, at most one BACKTRACK message is sent over each edge since the barrier variables prevent redundant backtracking. Thus, there are at most $|E|$ BACKTRACK messages per phase and $O(|V|^{1/2}|E|)$ messages for the algorithm.

During the search for augmenting paths, a GO message is sent from a bridge vertex to its buddy to signal the buddy to perform depth first search. One GO message is sent for each bridge. Since there are at most $|E|$ bridges and each bridge is discovered at most once during each phase, the number of GO messages is at most $|E|$ per phase. The total number of GO messages is $O(|V|^{1/2}|E|)$.

Next we consider the TAKINGDCV and CLAIMDCV messages. During each phase, there is at most one TAKINGDCV message associated with each bridge. Thus the number of TAKINGDCV messages for the algorithm is $O(|V|^{1/2}|E|)$.

During each phase, at most one CLAIMDCV message is sent over each edge. This is ensured by the barrier variables which keep track of the DCV. Thus the number of CLAIMDCV messages for the algorithm is $O(|V|^{1/2}|E|)$.

If the alternating depth first search finds a blossom, then the common vertex that is the base of the blossom sends BLOS messages which are forwarded by the intermediate vertices until they reach the bridge vertices. After the bridge vertices receive the BLOS messages, they send BLOSSOM messages to label the blossom. If there are no embedded blossoms, then each vertex of the blossom receives at most one BLOS message and sends at most one BLOSSOM message. If there are embedded blossoms, then a vertex of an embedded blossom may have to pass BLOSSOM messages for labeling outer blossoms. During each phase, there are no more than $|V|$ blossoms. Since each blossom is discovered and labeled once, each vertex receives no more than $|V|$ BLOS messages and sends no more than $|V|$ BLOSSOM messages. Thus the number of BLOS and BLOSSOM messages for the algorithm is $O(|V|^{5/2})$. Since there is one BLOSSOMREPLY message for each BLOSSOM message, the number of BLOSSOMREPLY messages for the algorithm is also $O(|V|^{5/2})$.

Each pair of bridge vertices discovering a blossom sends one LABELED message after the blossom is labeled. Since no more than $|V|$ blossoms are labeled during a phase, the total number of LABELED messages is $O(|V|^{3/2})$.

Only the vertices that are on an augmenting path send STARTINVERT messages. Note that if an augmenting path is found at the current search level, then the algorithm proceeds to the next phase. Since the augmenting paths discovered during each phase are disjoint and the vertices send STARTINVERT messages along augmenting paths, each vertex sends at most one STARTINVERT message. Thus, there are at most $|V|$ STARTINVERT messages during each phase and a total of $O(|V|^{3/2})$ STARTINVERT messages for the algorithm. Since there is one ENDINVERT message for each STARTINVERT message, the number of ENDINVERT messages for the algorithm is also $O(|V|^{3/2})$.

To determine the message complexity of the algorithm, we add up the total number of messages. Thus the message complexity of our distributed matching algorithm is $O(|V|^{5/2})$ messages.

4.3 Time Complexity

The time complexity of our distributed matching algorithm is the same as the message complexity. The reason is because the algorithm finds augmenting paths one at a time. During each phase, $O(|V|^2)$ messages are needed to find a maximal set of minimum length augmenting paths. Thus, the time complexity of the algorithm is $O(|V|^{5/2})$.

CHAPTER 5

MAXIMUM MATCHING ON TREES

We consider the performance of our distributed algorithm on trees. Matching on trees is much simpler since we do not need to consider blossoms or common vertices when performing depth first search. We show that our algorithm computes a maximum matching on trees using O($|V|$) messages.

Theorem 5.1: Given a tree T with root r, the distributed matching algorithm finds a maximum matching of T in one phase.

Before proving Theorem 5.1, we first present a sequential matching algorithm. Let $T(x)$ be the subtree of T rooted at x. Consider the following sequential algorithm called TREEMATCH, which takes one vertex x as input. Order the children $y_1, y_2, \cdots, y_n$ of x from left to right. TREEMATCH is called recursively on each child of x. We show that after the execution of TREEMATCH (x), $T(x)$ has a maximum matching. It follows that TREEMATCH (r) finds a maximum matching in T.

```
TREEMATCH ($x$)

if $x$ is a leaf then
        return

for $i$ = 1 to $n$
        call TREEMATCH ($y_i$)

if all children $y_i$ of $x$ are matched then
        leave $x$ exposed
else
        match $x$ with the leftmost child $y_i$ that is exposed
end
```

Suppose TREEMATCH (y_i) has been executed. Let $T(y_i)$ be the subtree of T rooted at y_i with a maximum matching. Suppose e is an exposed vertex distinct from y_i in $T(y_i)$. Let f be e's parent. Thus f is either y_i or a descendant of y_i.

Lemma 1: There is some child d of f matched with f.

Proof: Since TREEMATCH (y_i) has been executed, TREEMATCH must also have been executed on all descendants of y_i and in particular on f. Note that once a vertex is matched, it remains matched during the rest of the algorithm. Consequently, since e is exposed after the execution of TREEMATCH (y_i), e must have been exposed before the execution of TREEMATCH (f). Now consider the invocation of TREEMATCH (f). After TREEMATCH is called on each of f's children, the algorithm matches f with a child of f because at least one of f's children is exposed. Since e is exposed before the call TREEMATCH (f) and e remains exposed after the call, f must have been matched with a child d of f. □

Theorem 5.2: After the execution of TREEMATCH (x), $T(x)$ has a maximum matching.

Proof: We show by induction on the height of $T(x)$ that TREEMATCH finds a maximum matching. In the base case, x is a leaf. $T(x)$ is comprised of vertex x and no edges. For a graph of one vertex, the empty matching is maximum. Now assume the algorithm computes a maximum matching on subtrees with height less than the height of $T(x)$.

To prove $T(x)$ has a maximum matching, we show that $T(x)$ has no augmenting paths. Assume to the contrary that after the execution of TREEMATCH (x) there is an augmenting path P in $T(x)$. By the inductive hypothesis, P cannot be contained entirely within a subtree rooted at a descendant of x since any subtree of lesser height has a maximum matching. Thus P must contain vertex x. In addition, P must contain at least one exposed vertex e in $T(x)$ other than x.

We claim that at least one of the two exposed vertices of P is a descendant of a child y_i of x. We consider two cases.

Case 1: If x is exposed, then all children of x are matched. Hence the other exposed vertex e of P must be a descendant of some child y_i of x.

Case 2: If x is matched, then P must contain x and the child y_i matched with x. Thus at least one exposed vertex e of P must be a descendant of some y_i.

Since P is an augmenting path, there must be an alternating path leading from x to e. Let f be the parent of e, and let g be the parent of f. Vertex g exists since e is a descendant of a child of x. Note that g may be x. The edge (g, f) is free since by Lemma 1 f is matched with some child of f. The edge (f, e) is also free since e is exposed. Thus the path from x to e contains two consecutive free edges and is not an alternating path. Thus P is not an augmenting path. This is a contradiction. □

Proof of Theorem 5.1: When executed on trees, the method used to find augmenting paths in the distributed matching algorithm exhibits the same behavior as TREEMATCH. We compute a matching in T by executing one phase of the distributed matching algorithm on T. Note that the spanning tree for the distributed algorithm is exactly T. Since at the start of the algorithm no vertices are matched, the search for bridges at level 0 finds that every edge is a bridge.

The leader sends a STARTAUGMENT message to its leftmost child. A vertex x that is not a leaf forwards the STARTAUGMENT message to its children from left to right one at a time. Vertex x is matched with the first child y_i that is not matched with a child of y_i.

Now we compute a matching in T using TREEMATCH as follows. We first compute the leader of T using the preprocessing of the distributed algorithm. Then the root r of T is the leader L in the distributed algorithm. For each vertex x in T that is not a leaf, we order the children of x from left to right. After the execution of TREEMATCH (r), each vertex x is either matched with a child of x, matched with the parent of x, or exposed. We examine the three cases.

Case 1: If x is matched with a child of x, then x is matched with the leftmost child y_i of x that is not matched with a child of y_i. This is the same child y_i that x is matched with in the distributed algorithm because we sent the STARTAUGMENT messages in leftmost depth first search order.

Case 2: Let w be the parent of x. If x is matched with w, then x is the leftmost child of w that is not matched with a grandchild of w. Thus either x is a leaf or all of the children y_i of x are matched with a child of y_i. In the distributed algorithm, x is also matched with w because x is the leftmost child of w that is not matched with a grandchild of w.

Case 3: If x is exposed, then we consider the following subcases:

Subcase 3.1: If x is r, then all the children y_i of x are matched. Thus each y_i is matched with some child of y_i. In the distributed algorithm, each y_i is also matched with some child of y_i since each y_i searches for an augmenting path before x does.

Subcase 3.2: If x is not r, then w, the parent of x, is matched with some other child of w. Thus x was not the leftmost exposed child of w. In the distributed algorithm, w is also matched with some child of w other than x since x was not the leftmost exposed child of w.

Since the matching of each vertex and edge of T in the distributed algorithm and in TREEMATCH is the same, the matchings computed by the distributed algorithm and TREEMATCH are the same. Since the matching found by TREEMATCH is maximum, the matching found by the distributed algorithm in one phase is also maximum. Thus the distributed algorithm finds a maximum matching of T in one phase. □

We now determine the message complexity of the distributed algorithm when executed on trees. Note that during the phase, the breadth first search finds all the bridges in one search level. We assume that the algorithm knows *a priori* that the graph is a tree and halts after one phase.

Theorem 5.3: Given a tree T, the distributed matching algorithm finds a maximum matching in T using $O(|V|)$ messages.

Proof: We use the distributed depth first search algorithm of Awerbuch (1985b) to construct the spanning tree and select the leader. The number of messages required for the preprocessing is $O(|E|)$. But since the graph is a tree, $|E| = |V| - 1$. Thus the number of messages for the preprocessing is $O(|V|)$.

Now we consider the number of messages used by the algorithm. Since the algorithm finds a maximum matching at the first search level, all augmenting paths have length 1. Thus we do not need to worry about blossoms, common vertices, or backtracking, and can eliminate the associated messages.

We briefly consider the remaining messages. The messages used for synchronizing the algorithm, e.g., STARTPHASE, STARTBFS, BRIDGES, and STARTAUGMENT, are sent along the tree. Thus there are $O(|V|)$ of these messages per phase. The messages used to search for bridges and to find augmenting paths, e.g., BRIDGESEARCH, GO, and ERASED, may be sent along all edges. Thus there are $O(|E|) = O(|V|)$ of these messages per phase. Since by Theorem 5.1 the algorithm finds a maximum matching in one phase, the number of messages required to compute a maximum matching is $O(|V|)$. Thus the total number of messages required by the algorithm is $O(|V|)$. □

Suppose we have the tree shown in Figure 12a. Then the maximum matching computed by the distributed algorithm and TREEMATCH (A) is shown in Figure 12b.

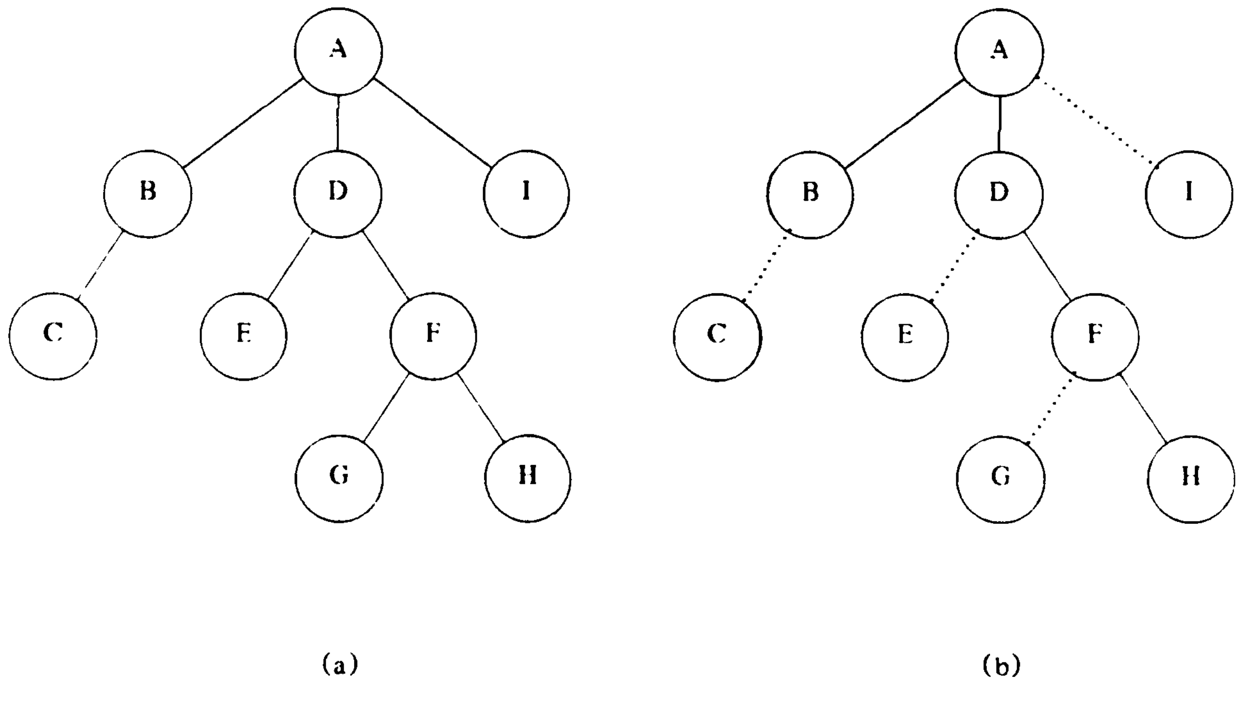


Figure 12. A Maximum Matching on a Tree

CHAPTER 6

CONCLUSIONS

We have presented a distributed algorithm for maximum cardinality matching in general graphs. In the worst case, our algorithm uses $O(|V|^{5/2})$ messages and requires $O(|V|^{5/2})$ time. We also showed that our algorithm finds a maximum matching in trees using only $O(|V|)$ messages.

We do not know if $O(|V|^{5/2})$ messages is the minimum number of messages required to compute a maximum matching in a distributed system. We could not construct a worst case example for our algorithm that requires $O(|V|^{5/2})$ messages. In fact, it seems that the total number of search levels required by the algorithm over all phases in the worst case is $O(|V|)$. Thus, if there is a more efficient method of dealing with blossoms and the above conjecture is true, then the message complexity of our algorithm can likely be reduced.

Other problems to consider are distributed algorithms for maximum weighted matching in both bipartite and general graphs. Galil *et al.* (1986) presented an $O(|E||V|\log|V|)$ sequential algorithm for finding a maximal weighted matching in general graphs. Galil (1986) surveyed some parallel algorithms that have been recently developed for maximum cardinality matching and maximum weighted matching. We know of no distributed algorithms for maximum weighted matching in either bipartite or general graphs.

REFERENCES

Awerbuch, B. (1985a), "Complexity of Network Synchronization," *J. ACM*, vol. 32, no. 4, pp. 804-823.

Awerbuch, B. (1985b), "A New Distributed Depth-First-Search Algorithm," *Inf. Proc. Let.*, vol. 20, no. 3, pp. 147-150.

Berge, C. (1957), "Two Theorems in Graph Theory," *Proc. Nat. Acad. Sci.*, vol. 43, pp. 842-844.

Eckstein, D. (1977), "Parallel Processing Using Depth-First-Search and Breadth-First-Search," Ph.D. Dissertation, Dept. of Computer Science, Univ. of Iowa, Iowa City, Iowa, 1977.

Edmonds, J. (1965), "Paths, Trees, and Flowers," *Can. J. Math.*, vol. 17, pp. 449-467.

Even, S. and Kariv, O. (1975), "An $O(n^{2.5})$ Algorithm for Maximum Matching in General Graphs," *Proc. of the 16th Annual IEEE Sym. on Foundations of Computer Science*, IEEE, pp. 100-112.

Gabow, H. (1976), "An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs," *J. ACM*, vol. 23, pp. 221-234.

Gafni, E., Loui, M., Tiwari, P., West, D., and Zaks, S. (1984), "Lower Bounds on Common Knowledge in Distributed Algorithms," Technical Report R-1017 (1984), Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

Galil, Z. (1986), "Efficient Algorithms for Finding Maximum Matching in Graphs," *Comp. Surveys*, vol. 18, no. 1, pp. 23-38.

Galil, Z., Micali, S., and Gabow, H. (1986), "An $O(EV \log V)$ Algorithm for Finding a Maximal Weighted Matching in General Graphs," *SIAM J. Computing*, vol. 15, no. 1, pp. 120-130.

Gallagher, R. (1982), "Distributed Minimum Hop Algorithms," Tech. Rep. LIDS-P-1175 (1982), M.I.T., Cambridge, MA.

Gallagher, R., Humblet, P., and Spira, P. (1983), "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Trans. on Programming Languages and Systems*, vol. 5, no. 1, pp. 66-77.

Hopcroft, J. E. and Karp, R. M. (1973), "An $n^{2.5}$ Algorithm for Maximum Matching in Bipartite Graphs," *SIAM J. Computing*, vol. 2, no. 4, pp. 225-231.

Kameda, T. and Munro, I. (1974), "A $O(|V||E|)$ Algorithm for Maximum Matching of Graphs," *Computing*, vol. 12, pp. 91-98.

Micali, S. and Vazirani, V. (1980), "An $O(\sqrt{|V|}|E|)$ Algorithm for Finding Maximum Matching in General Graphs," *Proc. 21st Annual IEEE Sym. on Foundations of Computer Science*, IEEE, pp. 17-27.

Papadimitriou, C. and Steiglitz, K. (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ.

Peterson, P. (1985). "The General Maximum Matching Algorithm of Micali and Vazirani." Technical Report T-163 (1985). Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

Schieber, B., and Moran, S. (1986). "Slowing Sequential Alogrithms for Obtaining Fast Distributed and Parallel Algorithms: Maximum Matchings." *Proc. of the 5th Annual ACM Sym. on Principles of Distributed Computing*, pp. 282-292.

Segall, A. (1983). "Distributed Network Protocols." *IEEE Trans. Inf. Theory*, vol. 29, pp. 23-25.

Shiloach, Y., and Vishkin, U. (1982). "An $O(n^2 \log n)$ Parallel MAX-FLOW Algorithm." *J. Algorithms*, vol. 3, pp. 128-146.

END

3- 87

DTIC